Proyecto fin de carrera

Ingeniería técnica en informática de gestión

Sistema para la monitorización en tiempo real de una arquitectura basada en eventos

Autor: Eduardo Posadas Fernández

Tutor: Álvaro Luis Bustamante

Co-director: Enrique David Martí Muñoz

Colmenarejo, 29 de Octubre de 2015

Título: Sistema para la monitorización en tiempo real de una arquitectura basada en eventos.

Autor: Eduardo Posadas Fernández

Co-director: Enrique David Martí Muñoz

EL TRIBUNAL

Presidente: _____

Vocal: _____

Secretario: _____

Realizado el acto de defensa y lectura del Proyecto Fin de Carrera el día __ de _____ de 20__ en Leganés, en la Escuela Politécnica Superior de la Universidad Carlos III de Madrid, acuerda otorgarle la CALIFICACIÓN de

VOCAL

SECRETARIO PRESIDENTE

Índice general

Índice de ilustraciones

1 Introducción

En la actualidad la sociedad depende de sistemas informáticos de elevada complejidad. A lo largo de la vida diaria se es usuario, consciente o inconscientemente, de multitud de complejos sistemas controlados y supervisados por software. Multitud de infraestructuras dependen del buen funcionamiento del software que las controla y se asume que nunca fallan. Grandes organizaciones, ya sean empresas u otra clase de organismos gubernamentales o privados, apoyan su funcionamiento diario en software plenamente confiadas en él. Sistemas de inventariado de grandes almacenes, sistemas de gestión de logística, sistemas de control de redes eléctricas, de redes de telecomunicaciones, sistemas de control de tráfico, etc... toda clase de sistemas, esenciales o simplemente de entretenimiento, controlados y supervisados por software dan servicio a la sociedad. El fallo de uno de estos sistemas puede acarrear consecuencias de toda clase, desde mala imagen para la organización propietaria o gestora hasta consecuencias fatales.

La dependencia de la sociedad actual del software es clara y sin embargo, a pesar de las diferentes metodologías de desarrollo y la abundante literatura sobre la ingeniería del software, los desarrollos siguen teniendo fallos o incluso siguen sin cumplir los requerimientos. Ya sea por algún malentendido en una entrevista con el cliente, una comparación fallida de un número en coma flotante bajo ciertas circunstancias, una interacción no prevista entre dos hilos de ejecución... la posibilidad de error sigue ahí.

Evidentemente, no hay que caer en el catastrofismo. Gracias al software creado en las últimas décadas se han conseguido avances importantes y la repercusión que pueda tener la integración de los sistemas controlados por software en la vida cotidiana aún no se conoce con totalidad, ya que es un proceso que está muy lejos de su fin. Los sistemas computarizados están presentes tanto en el domicilio particular como en las grandes empresas e instituciones. Cuando alguien realiza un gesto cotidiano como ver en la televisión un noticiario o desplazarse en automóvil al trabajo no es consciente de la cantidad de software que está usando y que se ha usado para hacer que esos productos y servicios estén a su disposición, desde el software de la propia televisión o del propio coche, el tratamiento de vídeos, el software para CAD, servicios de telecomunicaciones... todo funciona como debería, y lo hace así rutinariamente.

A pesar de esto se hace evidente, sobre todo en sistemas críticos [1] que van a estar en uso durante largos periodos de tiempo, la necesidad de hacer un seguimiento y supervisión del funcionamiento del software para conseguir una evolución positiva y acorde con los resultados deseados. Durante este periodo de explotación y mantenimiento se ha de implementar, en los sistemas en los que el coste del posible fallo lo justifique, un sistema de supervisión que dé información fidedigna y útil para la corrección de los posibles fallos, a ser posible, de forma anticipada.

1.1 Motivación

En un trabajo de fin de grado anterior [2] se creó un *framework* para el procesado de datos genéricos de forma paralela y muy adaptable. Los requerimientos principales de este desarrollo fueron que el procesado de los datos pudiese ser hecho de forma paralela para poder maximizar el rendimiento y que fuese flexible y generalista para permitir su uso en una múltiple variedad de desarrollos.

El *framework* permite definir diferentes unidades de procesamiento que pueden ser adaptadas para que realicen las operaciones necesarias para la resolución de un problema propuesto. Estas unidades de procesamiento pueden recibir los datos de entrada de múltiples fuentes externas o de otras unidades de procesamiento y, de la misma manera, pueden enviar la salida de su procesamiento a otras unidades de procesamiento o a los destinos externos que se le especifiquen. Además, las diferentes unidades de procesamiento que se definan pueden realizar sus operaciones de forma secuencial o de forma paralela, según se establezca en base a un criterio de tiempo o por la recepción de un evento.

Este abanico de posibilidades permite crear una red de unidades de procesamiento en la que cada una de ellas trabaja en una parte en concreto de la resolución del problema propuesto, esperándose en algunos casos las unas a las otras, realizando los procesamientos en paralelo o secuencialmente y pasándose entre ellas los diferentes resultados de sus procesamientos hasta producir el resultado final.

Los desarrollos implementados usando este *framework* pueden alcanzar una complejidad considerable. Esto unido a que durante la ejecución de las aplicaciones basadas en este *framework* las unidades de procesamiento cambian de estado constantemente y realizan un continuo intercambio de información entre ellas conlleva que la depuración de las aplicaciones no sea sencilla. Por esto se hace necesario que exista una aplicación capaz de mostrar gráficamente y en tiempo real la evolución de los estados en los que se encuentran las unidades de procesamiento y la variación de las relaciones entre ellas.

Con esta representación gráfica se espera obtener una visión global y en tiempo real del funcionamiento del *framework* posibilitando la resolución de una forma más eficaz de los problemas que pudieran presentarse en las aplicaciones que usen el *framework*.

1.2 Objetivo

El objetivo es el desarrollo de una aplicación multiplataforma capaz de mostrar en una interfaz gráfica y en tiempo real el estado y las relaciones de las diferentes unidades de procesamiento de un *framework* basado en eventos.

La comunicación con el *framework* se realizará mediante una conexión TCP, siendo el *framework*

el encargado de enviar a la interfaz gráfica los distintos cambios de los estados o de las relaciones entre unidades de procesamiento. La interfaz gráfica variará en tiempo real la apariencia de los elementos mostrados en base a la información recibida.

El protocolo usado para la comunicación entre la aplicación gráfica y el *framework* hará uso del estándar JSON por lo que, si en un futuro se requiriese, sería posible ampliarlo fácilmente.

Con la finalidad de evaluar la idoneidad, rendimiento y funcionalidad de la aplicación desarrollaba, esta será usada para visualizar gráficamente la evolución de las unidades de procesamiento de un sistema de vigilancia marítima que ha sido desarrollado usando el *framework* descrito brevemente en el apartado anterior y que se describe con más detalle en el capítulo 3.

2 Estado del arte

Como se ha mencionado en la introducción, en los sistemas donde el coste de un posible fallo lo justifique se hace indispensable el uso de sistemas de monitorización para posibilitar la mejora del propio sistema y supervisar su correcto funcionamiento. Es por esto por lo que existen soluciones anteriores que se han evaluado para comprobar si podían ser aplicados para la supervisión del *framework*.

Un primer ejemplo relevante de uso generalizado de sistemas de supervisión para monitorización de procesos son los los sistemas SCADA. Estos sistemas son usados para la supervisión y control a distancia de los procesos industriales. Usando estos sistemas se supervisan y controlan en las factorías industriales los diferentes pasos que se llevan a cabo para realizar la transformación de las materias primas en el producto final. Cada uno de estos pasos, además de ser supervisado, recibe correcciones en base a las mediciones realizadas con anterioridad, lo que posibilita que el funcionamiento sea totalmente automático o que apenas sean necesarias intervenciones manuales. Además de la automatización, estos sistemas permiten la modificación a distancia de los diferentes pasos del proceso, posibilitando un completo control de la producción.

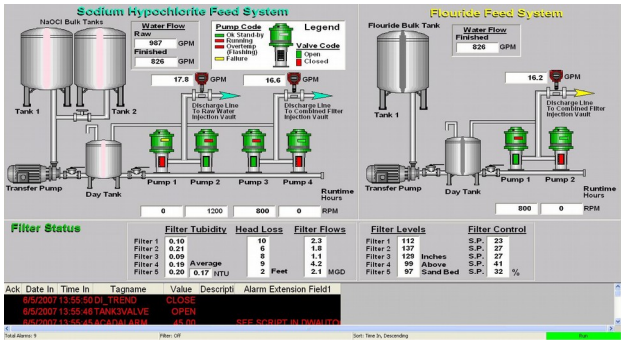

Ilustración 1: Interfaz sistema SCADA

Adicionalmente, los sistemas SCADA mantienen un registro histórico de los datos recolectados por la supervisión. Esto permite la elaboración de previsiones en base a estos datos y, en muchas implementaciones, el traspaso de esta información a los sistemas ERP de la compañía para la toma de decisiones de negocio.

Los sistemas SCADA, son ampliamente usados y existe abundante literatura sobre ellos. La implementación de sistemas SCADA en los procesos industriales tiene como objetivo principal la mejora del proceso mediante el ahorro de costes, la eliminación de cuellos de botella, el control del proceso para conseguir adaptabilidad a la demanda del mercado, estandarización de los elementos de supervisión con la consiguiente independencia de proveedores, etc.

La principal traba para su aplicación a la monitorización del *framework* es que los sistemas SCADA están diseñados para entornos industriales siendo muy dificultosa su implementación en sistemas controlados por software. La entradas suelen provenir de dispositivos físicos llamados PLC situados en la planta industrial y su interfaz gráfica no está pensada para reflejar los cambios de estado de las unidades de procesamiento de un *framework*.

Otro ejemplo de sistemas de monitorización ampliamente usados son los sistemas de monitorización de redes. Estos sistemas se usan para la supervisión de redes de computadoras y los elementos que las interconectan. El sistema de monitorización puede recibir datos de un agente instalado en el nodo o desde el propio nodo mediante algún protocolo como SNMP o NetFlow. Habitualmente se instalan agentes en los ordenadores, que tienen la posibilidad de ejecutar programas, y se reciben datos por SNMP o NetFlow de los elementos de interconexión de la red, que no suelen ofrecer la posibilidad de ejecutar un programa. Estos sistemas de monitorización generalmente tienen la capacidad de poder realizar mapas lógicos de red lo que facilita la gestión de la misma. Al igual que los sistemas SCADA, los sistemas de monitorización de redes guardan en bases de datos los datos recolectados de lo nodos de integrantes de la red para permitir hacer previsiones.

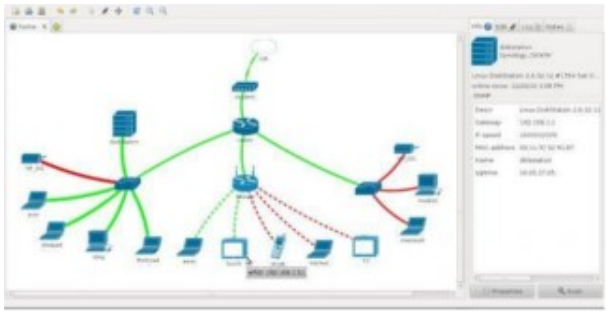

Ilustración 2: Mapa de red

También permiten la creación de alarmas en base a los datos obtenidos. Estas alarmas pueden desencadenar multitud de acciones, como pueden ser notificaciones por diferentes medios o la ejecución de acciones en los nodos de la red que estén ejecutando un agente de monitorización.

Como inconvenientes para la aplicación de un sistema de monitorización de redes para la monitorización del *framework* es de destacar que, como primer obstáculo, se debería implementar en el *framework* un cliente de un protocolo conocido por el sistema de monitorización, como SNMP, e implementar usando este protocolo las primitivas necesarias. Los protocolos de monitorización de red están diseñados para enviar datos numéricos y sería difícil enviar otro tipo de valores, como listas de valores alfanuméricos, es decir estos protocolos son poco flexibles.

Otra desventaja de elegir un sistema de monitorización de redes para monitorizar el *framework* es que los mapas de red, que en este caso serían una representación gráfica de las unidades de procesamiento, no son fácilmente manipulables ni se adaptan a las necesidades especificas de la depuración del *framework*.

Es por esto que, después de analizar las alternativas, se decidió hacer un desarrollo *ad hoc* dada el especificidad del problema planteado y la imposibilidad de encontrar una solución ya desarrollada.

Para abordar el requerimiento de que la aplicación fuera multiplataforma se evaluaron diversas bibliotecas de controles gráficos, como JavaFx o wxWidgets, siendo elegida finalmente Qt. Las razones por los que esta biblioteca fue la elegida fueron:

- Es una biblioteca madura que ha sido ampliamente utilizada para el desarrollo de aplicaciones multiplataforma con anterioridad. Entre las aplicaciones que la usan aparecen algunas muy conocidas como Virtualbox o Google Earth.
- Existe multitud de código disponible que puede ser estudiado e incluso usado. El disponer de abundante código y ejemplos de uso facilita enormemente el aprendizaje y los futuros desarrollos.
- La documentación es excelente y accesible directamente desde el entorno de desarrollo. Esto, como el punto anterior, facilita muchísimo el desarrollo y posibilita que el aprendizaje sea rápido y productivo.
- El aspecto de la aplicación desarrollada es el propio de cada sistema operativo lo que no produce en el usuario la sensación de estar usando una aplicación extraña.
- Además de los controles gráficos, Qt incluye otras clases que dan otras funcionalidades multiplataforma como una abstracción de los *sockets* TCP, una API para guardar la configuración de la aplicación o un analizador sintáctico JSON.

3 Descripción del *framework* de procesado de información

En el presente capítulo se describe el *framework* [2] de procesado de información del que la aplicación gráfica mostrará sus componentes, la unidades de procesamiento, y las relaciones entre ellas.

3.1 Introducción

El *framework* tiene como objetivo el procesado de forma paralela de datos proveniente de diferentes fuentes y su posterior envío. En su diseño no se tuvo en cuenta ningún propósito en concreto, al contrario se diseñó para poder ser usado para resolver problemas de muy diferente índole, es decir es un *framework* de propósito generalista.

Con estos objetivos en mente, el *framework* da la posibilidad de definir diferentes entidades de procesamiento. Estas entidades pueden ser configuradas de diferentes formas en cuanto al procesamiento a efectuar y al tratamiento de las entradas y de las salidas.

Los datos de entrada pueden provenir de distintas fuentes, ya sean externas o internas. Las fuentes externas pueden ser sensores de cualquier tipo, ficheros, bases de datos, datos recibidos vía red, datos introducidos por el usuario... Los datos provenientes de otra entidad de procesamiento se consideran de origen interno. Un entidad puede estar especializada en recibir datos de distintas fuentes simultáneamente.

Tras haber recibido los datos, la entidad realiza el procesamiento de esta información dependiendo de la forma que en que esté configurada. El tratamiento se puede realizar secuencialmente, en base a un evento o en base a una planificación temporal. Con esto se consigue que el *framework* sea lo suficientemente flexible como para adaptarse a cualquier situación, por ejemplo es posible que el procesado de unos datos en concreto deba realizarse por lotes, pero que el procesamiento de otro tipo de datos deba ser secuencial.

En el caso de que el tipo de procesamiento elegido no sea el secuencial, el *framework* se encarga de crear un nuevo hilo de ejecución que realizará el procesamiento en el momento adecuado. De esta forma se evitan posibles bloqueos y se aumenta el rendimiento del sistema al realizarse de forma paralela la recepción, el procesado y el envío.

Una vez realizado el procesamiento, la entidad se encarga del envío de los datos. Este envío se puede realizar, al igual que la recepción, de forma secuencial, en base a un evento o en base a una planificación temporal. El *framework* también ofrece la posibilidad de enviar los datos a diferentes destinos de forma simultanea, ya sea a otra entidad o a un destino externo al sistema.

Se muestra en la siguiente ilustración una configuración de ejemplo del sistema de procesado con

múltiples entidades de procesamiento que reciben entradas tanto externas como internas.

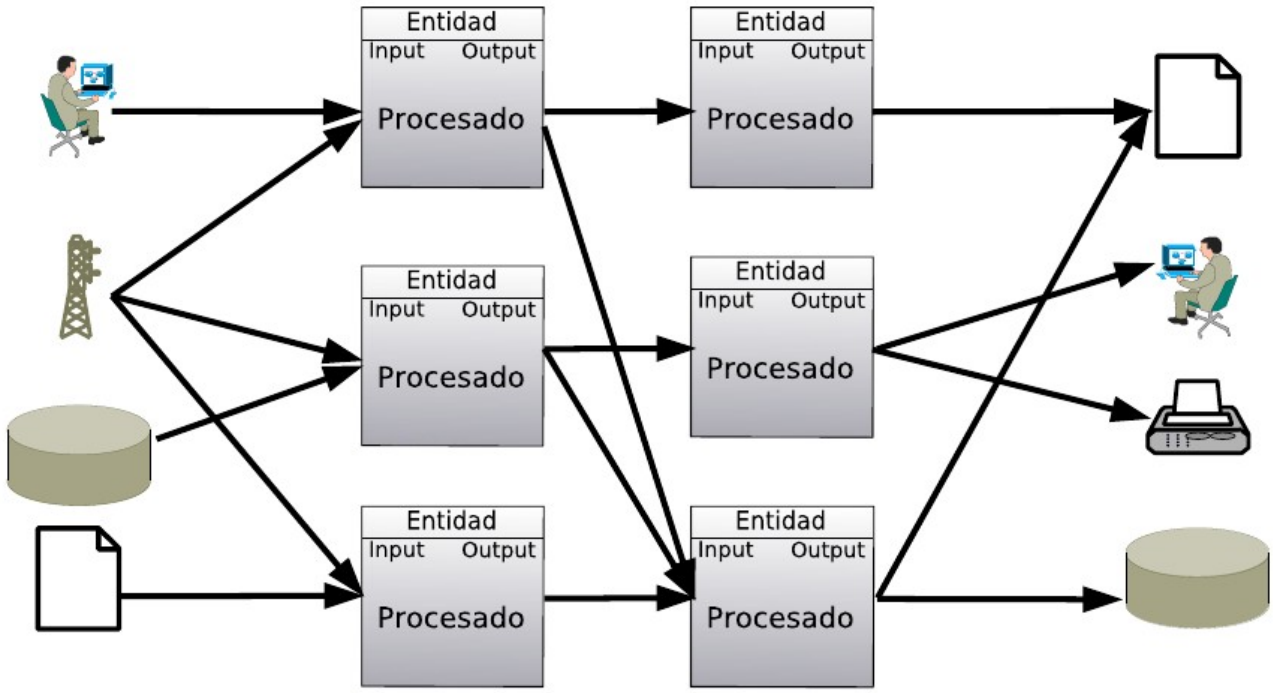

Ilustración 3: Ejemplo de uso del framework

Como se puede observar el sistema ofrece la flexibilidad adecuada para realizar el procesamiento de datos provenientes de distintas fuentes y enviar las salidas a múltiples dispositivos.

3.2 MessageProcessor

El *MessageProcessor* es el nombre que se le ha dado a las entidades de procesamiento del *framework* mencionadas anteriormente. Es la unidad sobre la que se basa el funcionamiento del *framework* y sobre la que se actuará si se utiliza el *framework*.

Su funcionamiento se puede dividir en cinco fases:

- Entrada de datos: el *MessageProcessor* puede recibir datos de una fuente externa, interna o los puede generar él mismo.
- Gestión de los datos de entrada: existen cuatro formas de gestión de los datos de entrada que se detallarán a continuación. Si los datos no son entregados inmediatamente para su procesado, el *MessageProcessor* inicia automáticamente un nuevo hilo de ejecución que se encarga de enviar los datos para el procesado en el momento adecuado dependiendo de la configuración establecida.
- Procesado: es la fase que se encarga de hacer la lógica de la operación. Toda transformación

que se produce en los datos se produce en esta fase del *MessageProcessor*.

- Gestión de los datos de salida: al igual que en la gestión de los datos de entrada, existen cuatro formas de gestión de los datos de salida. Y de la misma manera, si los datos no son enviados inmediatamente será en este paso en el que se genere un nuevo hilo de ejecución para determinar cuando se envían los datos.
- Envío de datos: en esta fase se envían los datos a todas las entidades que estén relacionadas con este *MessageProcessor*.

3.2.1 Entrada de datos

El *MessageProcessor* tiene como primera fase la entrada de datos que consiste en la recepción propiamente dicha de los datos que posteriormente serán procesados. Como se ha comentado con anterioridad, la entrada de datos puede ser interna, si los datos provienen de otro *MessageProcessor*, o externa, si los datos provienen de una fuente externa al sistema.

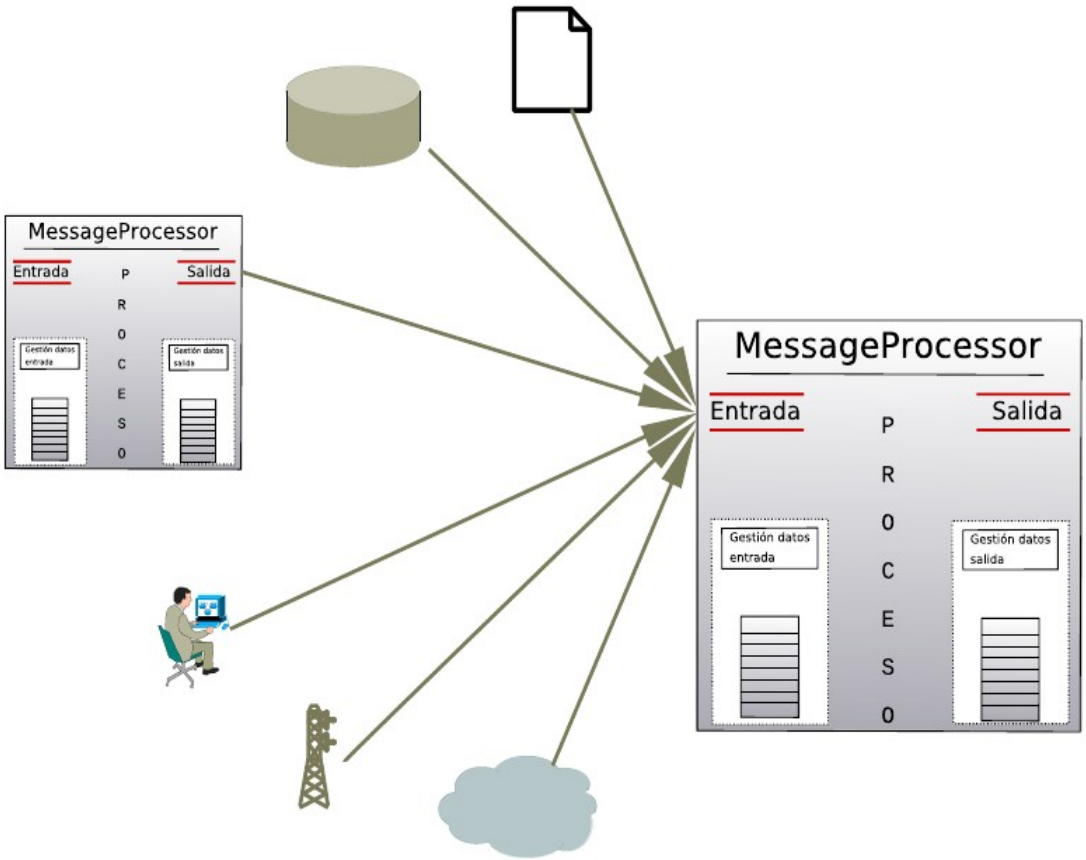

Ilustración 4: Entrada de datos a un MessageProcessor

Para poder adaptarse a las posibles fuentes de datos externas, los *MessageProcessor* ofrecen la posibilidad de ser configurados para poder interpretar los datos provenientes del exterior y así conseguir un punto de entrada de datos a la aplicación.

3.2.2 Gestión de los datos de entrada

Una vez los datos han sido recibidos, el *MessageProcessor* tiene la posibilidad de realizar el paso de los datos a la fase de procesado de cuatro formas: síncronamente, asíncronamente, por eventos y por tiempo.

La diferencia fundamental es que en el modo de funcionamiento síncrono los datos se pasan directamente a la fase de procesado, mientras que en los modos de funcionamiento asíncrono, por eventos y por tiempo se almacenan en memoria y se lanza un hilo de ejecución que se encarga de pasar los datos a la fase de procesado cuando las necesidades del problema así lo requieran. En estos modos de funcionamiento se consigue que el hilo que está realizando la recepción de datos no se bloquee mientras se está produciendo el procesamiento de los datos.

3.2.2.1 Gestión de datos de entrada síncrona

En este modo de gestión de datos de entrada es el propio *MessageProcessor* el que se encarga de pasar los datos a la fase de procesado sin utilizar ningún tipo de copia intermedia ni de planificación.

Este modo de gestión de datos de entrada está pensado para situaciones en las que el procesado de los datos no sea un cuello de botella, permitiendo así que el *MessageProcessor* sea capaz de enviar los datos de salida tan rápido como llegan los datos de entrada.

Debido a que es el propio hilo de ejecución del *MessageProcessor* el que realiza el paso de datos a la fase de procesado, no se pueden recibir nuevos datos hasta que haya terminado el procesado del dato anterior.

3.2.2.2 Gestión de datos de entrada asíncrona

En este modo de gestión de datos de entrada se lanza un nuevo hilo de ejecución que se encarga de encolar en memoria los datos recibidos mientras se está realizando el procesamiento de datos recibidos con anterioridad.

Con esto se consigue que la recepción de datos no se bloquee y permite al *MessageProcessor* continuar recibiendo datos mientras está realizando el procesado de datos anteriores al delegar en un nuevo hilo de ejecución la recepción y el encolamiento de los datos hasta que puedan ser procesados.

3.2.2.3 Gestión de datos de entrada por eventos

En este modo de gestión de datos de entrada se lanza un nuevo hilo de ejecución que, al igual que en el anterior modo, se encarga de encolar en memoria los datos recibidos. A diferencia con el anterior modo, el paso de los datos a la fase de procesado se realiza cuando se produce un evento especificado en la configuración del *MessageProcessor*.

Con este tipo de gestión de datos de entrada se consigue que el procesado de un grupo de datos de entrada sea realizado en conjunto basándose en un evento recibido por el *MessageProcessor*. Por ejemplo, en el caso del uso del *framework* para la implementación de un sistema de vigilancia marítima se puede usar para agrupar todas las detecciones recibidas en una misma vuelta de radar, siendo el evento el paso por norte del sensor.

3.2.2.4 Gestión de datos de entrada por tiempo

Este modo de gestión de datos de entrada es similar al anterior, excepto que el paso de los datos a la fase de procesamiento se realiza periódicamente, no en base a un evento.

3.2.3 Procesamiento de los datos

Esta es la fase que da funcionalidad al *MessageProcessor*. La suma de las implementaciones de las fases de procesamiento de los diferentes *MessageProcessor* es la implementación de la lógica del sistema.

Mientras que las otras fases del *MessageProcessor* se dedican a la gestión de la entrada y salida de los datos, esta fase es la encargada de realizar la lógica del procesado de datos.

3.2.4 Gestión de los datos de salida

Al igual que para los datos de entrada, el *MessageProcessor* provee de cuatro configuraciones posibles para la gestión de los datos de salida. El objetivo es dar la posibilidad al desarrollador de elegir la forma en la que los datos serán enviados.

Para la gestión de los datos de salida el *framework* también provee los modos síncrono, asíncrono, por eventos y por tiempo. Análogamente a la gestión de los datos de entrada, en la configuración síncrona los datos se envían según se procesan mientras que en el resto de los modos se crea un hilo que almacena en memoria los datos procesados hasta su envío.

Es destacable que el modo de gestión de los datos de salida es independiente del modo de gestión de los datos de entrada.

3.2.4.1 Gestión de datos de salida síncrono

En este modo de gestión de los datos de salida es el propio hilo de ejecución del *MessageProcessor* el que realiza el envío del dato una vez terminado el procesamiento. Esto puede ocasionar bloqueos,

por lo que este modo sólo se utiliza en casos muy concretos.

3.2.4.2 Gestión de datos de salida asíncrono

Al igual que en la entrada, en este modo de gestión de datos de salida el *MessageProcessor* lanza un nuevo hilo de ejecución que se encarga de encolar en memoria los datos procesados que están listos para el envío, liberando al hilo de ejecución de la tarea de la entrega de los datos de salida.

Esto evita que se puedan producir bloqueos y proporciona al *MessageProcessor* mayor rendimiento ya que el hilo de ejecución principal puede seguir con el procesamiento de datos mientras delega en otro el envío.

3.2.4.3 Gestión de datos de salida por eventos

Al igual que el anterior modo de gestión de los datos de salida, en este modo el *MessageProcessor* también lanza otro hilo de ejecución que se encarga de la gestión de los datos procesados. La diferencia es que el envío de los datos se produce con la activación de un evento especificado en el configuración del *MessageProcessor*.

Con este modo de gestión de la salida de datos se consigue que el envío de datos se produzca por lotes y en base al disparo de un suceso, lo que puede ser algo requerido para la resolución de algunos problemas o por simple eficiencia.

3.2.4.4 Gestión de datos de salida por tiempo

Este modo de gestión de datos de salida es análogo al anterior. La diferencia es que el envío se realiza periódicamente y no en base a un evento.

También se lanza otro hilo de ejecución que se encarga del envío lo que, como se ha dicho, evita la aparición de bloqueos.

3.2.5 Envío de los datos

En esta fase es en la que se envían los datos ya procesados a todos los destinos que estén conectados con el *MessageProcessor*. El envió, al igual que la recepción, se puede realizar a un destino externo o interno, es decir a otro *MessageProcessor* o a un destino externo al sistema, como una base de datos, un fichero, una impresora, a la pantalla...

Es importante destacar que el envío se realiza a todos las entidades que estén conectadas con el *MessageProcessor*, siendo la entidad destinataria la encargada de decidir si debe tratar ese dato o no.

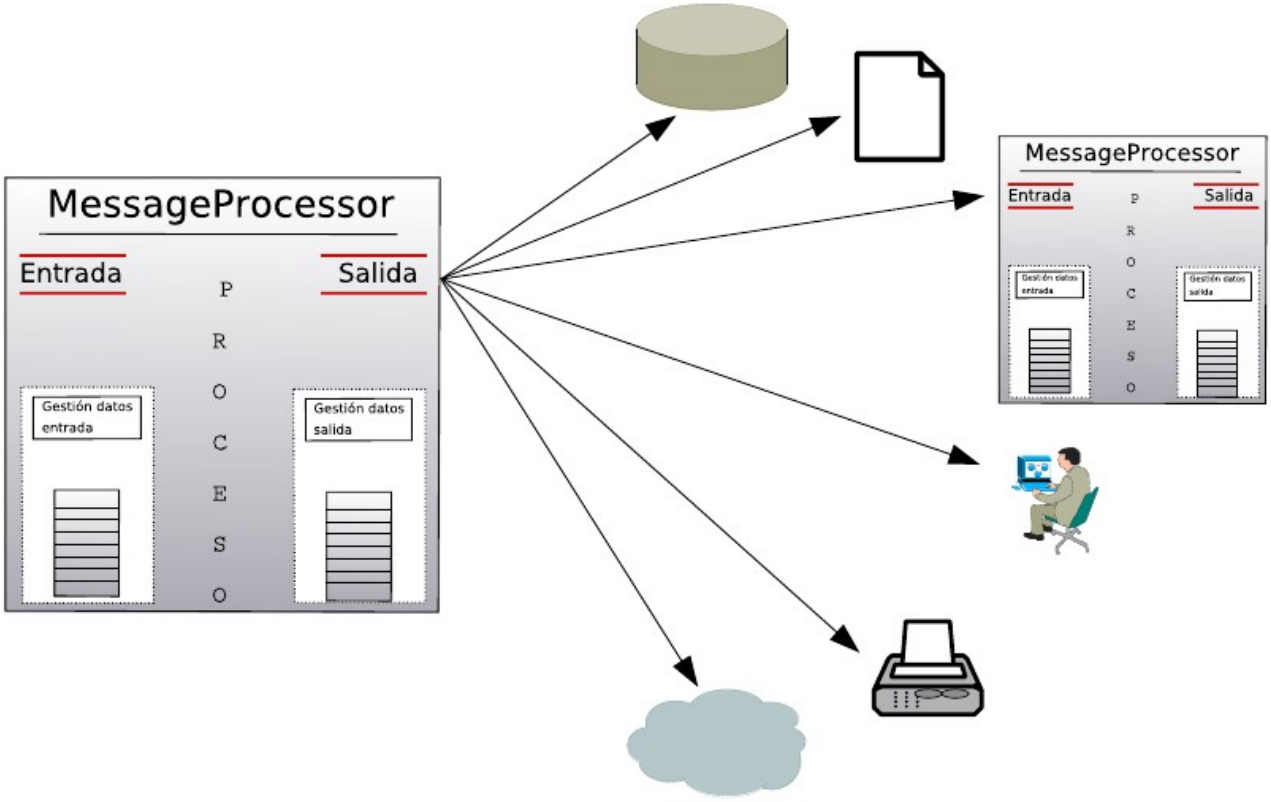

Ilustración 5: Salida de datos de un MessageProcessor

4 El sistema de monitorización

Una vez descrito en el capítulo anterior el *framework* de procesamiento de la información se describe en el presente capítulo la aplicación desarrollada para su monitorización.

Este capítulo se divide en cuatro apartados. En el primero se presenta la planificación y el presupuesto del proyecto. En el segundo se realiza una descripción de la apariencia visual del sistema de monitorización y de las diferentes posibilidades que da la aplicación al usuario. El siguiente detalla el protocolo de comunicación entre el *framework* y el sistema de monitorización. El último explica el diseño interno de la aplicación, explicando las diferentes clases que se han creado y sus interacciones.

La aplicación desarrollada presenta en pantalla y en tiempo real el estado de los diferentes *MessageProcessor* y las relaciones entre ellos. Para conseguir esto, el *framework* manda el estado de los diferentes *MessageProcessor* y sus relaciones al sistema de monitorización. Este se encarga de presentar la información en la pantalla y permanece a la espera de que el *framework* envíe los cambios en el estado y en las relaciones que los *MessageProcessor* tiene establecidas entre ellos.

La comunicación entre el *framework* y la aplicación de monitorización se realiza mediante una conexión TCP y usando JSON como protocolo para el intercambio de información. Es destacable que la aplicación de monitorización permanece a la escucha pasivamente, es decir actúa como servidor, y es el *framework* el que envía los cambios que se producen en los estados y relaciones de los *MessageProcessor*. La aplicación de monitorización se encarga de guardar esta información cuando se cierra y de volver a leerla cuando se arranca para evitar que el *framework* tenga que volver a enviar de nuevo toda la información.

La aplicación de monitorización informa al usuario de las acciones y los errores que puedan suceder mediante la barra de estado o mediante el cuadro de texto de la interfaz gráfica . En este cuadro de texto va quedando registro de los sucesos de interés con la hora a la que ocurren.

4.1 Planificación y presupuesto

En este apartado se detallarán la planificación seguida para la realización del presente proyecto así como los diferentes gastos en los que se ha incurrido y el presupuesto resultante.

4.1.1 Planificación

Para la realización del proyecto se han llevado a cabo las tareas de análisis del problema, diseño de la solución propuesta, desarrollo de la aplicación, pruebas de la aplicación, integración de la aplicación desarrollada con el *framework*, correcciones de la aplicación y pruebas finales. Alrededor

de la mitad de la fase de desarrollo se empezó a crear la documentación presentada. La duración en días de las diferentes fases del proyecto se muestra en la siguiente ilustración.

Nombre	Fecha de inicio	Fecha de fin	Duración
Análisis	1/10/14	21/10/14	15
Diseño	22/10/14	12/11/14	15
Desarrollo	13/11/14	10/04/15	100
Pruebas	13/04/15	11/05/15	20
Integración	12/05/15	2/06/15	15
Correcciones Desarrollo	12/05/15	2/06/15	15
Pruebas integración	3/06/15	17/06/15	10
Documentación	28/01/15	15/07/15	115

Ilustración 6: Planificación del proyecto

En base a estas tareas el diagrama de Gantt resultante es:

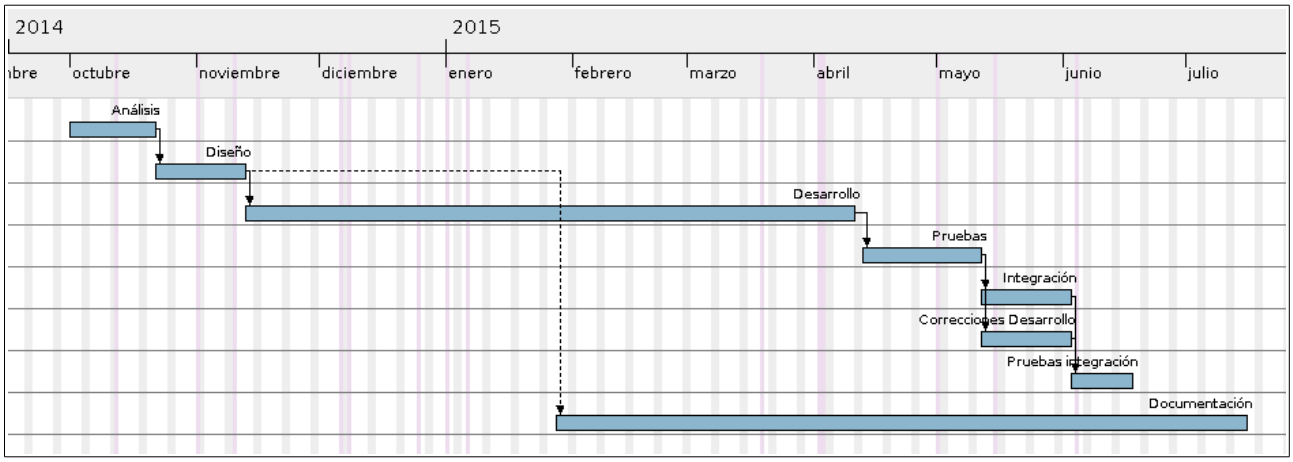

Ilustración 7: Diagrama de Gantt de la planificación del proyecto

4.1.2 Presupuesto

El desglose de los gastos[1] que ha supuesto la elaboración de este proyecto son:

1. Coste del personal

   Para los cálculos del coste de personal se ha tenido en cuenta que no ha habido una

1 Los gastos descritos aquí se especifican sin IVA ni otros impuestos.

dedicación exclusiva a la realización del proyecto por lo que se muestra en la tabla el porcentaje de la jornada laboral dedicado. Según la planificación mostrada en el apartado anterior y teniendo en cuenta estos porcentajes:

Categoría	Dedicación (días)	%Dedicación	Coste[2] en €/mes	Coste en €
Analista	30	50%	2487,44	1243,72
Desarrollador	100	50%	2487,44	4104,28
Probador	45	50%	2487,44	1865,58
Documentador	115	25%	2487,44	2363,07

Tabla 1: Desglose de los costes de personal

2. Coste del Hardware

   Para la realización del proyecto se ha usado hardware ya amortizado completamente con una antigüedad de alrededor de 6 años [4] a excepción de un ordenador portátil Apple MacBook Pro comprado por 592€.

3. Coste de las licencias de software

   La gran mayoría del software usado ha sido software libre o gratuito con la excepción de los sistemas operativos MS-Windows y OS X 10.9.

   La licencia de MS-Windows venía incluida en uno de los ordenadores portátiles usados y que, como se ha mencionado en el punto anterior, no se puede imputar como gasto al corresponder a hardware ya amortizado.

   La licencia de OS X venía incluida en la compra del portátil Apple MacBook Pro.

4. Costes indirectos

   Durante la realización del proyecto se ha incurrido en gastos que no son atribuibles directamente al proyecto como tal, como pueden ser el alquiler del despacho, el consumo eléctrico o la conexión a internet. Estos costes se han estimado en un 20% del total del coste del proyecto.

Con estos gastos el presupuesto total es:

2 Como referencia para estimar los costes del personal se ha tomado el coste laboral medio por trabajador en el primer trimestre del año 2015 publicado por el INE [3]

Concepto	Coste
Coste personal	9576,65 €
Coste hardware	592,00 €
Coste software	0 €
Costes indirectos	1915,33 €
Coste total	12083,98 €
Coste total con IVA (21 %)	14621,62 €

Tabla 2: Resumen de costes

El presupuesto total de este proyecto asciende a la cantidad de CATORCE MIL SEISCIENTOS VEINTIUN EUROS CON SESENTA Y DOS CÉNTIMOS.

4.2 Descripción de la interfaz gráfica

4.2.1 Ventana principal

La imagen de la ilustración 8 muestra una captura de la ventana principal de la aplicación con una serie de *MessageProcessor*. Como se puede observar la ventana principal está dividida en cuatro zonas, de izquierda a derecha y de arriba a abajo:

- Lista de *MessageProcessor*: se ofrece una lista de todos los *MessageProcessor* que están en la parte gráfica. Al pinchar en un elemento de la lista el visor gráfico se centra en el *MessageProcessor* seleccionado.
- Datos del *MessageProcessor*: muestra información del *MessageProcessor* seleccionado. Los datos mostrados son:
  - UID: entero que identifica unívocamente al *MessageProcessor* en el sistema.
  - Nombre: identificador descriptivo del *MessageProcessor*.
  - Entrada: modo en el que el *MessageProcessor* está gestionando los datos de entrada.
  - Salida: modo en el que el *MessageProcessor* está gestionando los datos de salida.
  - Conexiones de entrada: lista de *MessageProcessor* desde los que el *MessageProcessor* seleccionado esta recibiendo datos.
  - Conexiones de salida: lista de *MessageProcessor* a los que el *MessageProcessor* seleccionado esta enviando datos.

- Información adicional: parejas formadas por los elementos atributo y valor que son característicos del *MessageProcessor*.

- Representación gráfica: en esta zona se puede visualizar gráficamente los diferentes *MessageProcessor* y sus relaciones. Los *MessageProcessor* se pueden mover arrastrándolos con el ratón para conseguir el diagrama deseado.

- Registro: en este cuadro de texto quedan registrados errores de comunicación y sucesos relevantes, como pueden ser conexiones, desconexiones, errores de formato en una comunicación... Además de en este cuadro de texto, estos mensajes también se muestran en la barra de estado de la aplicación.

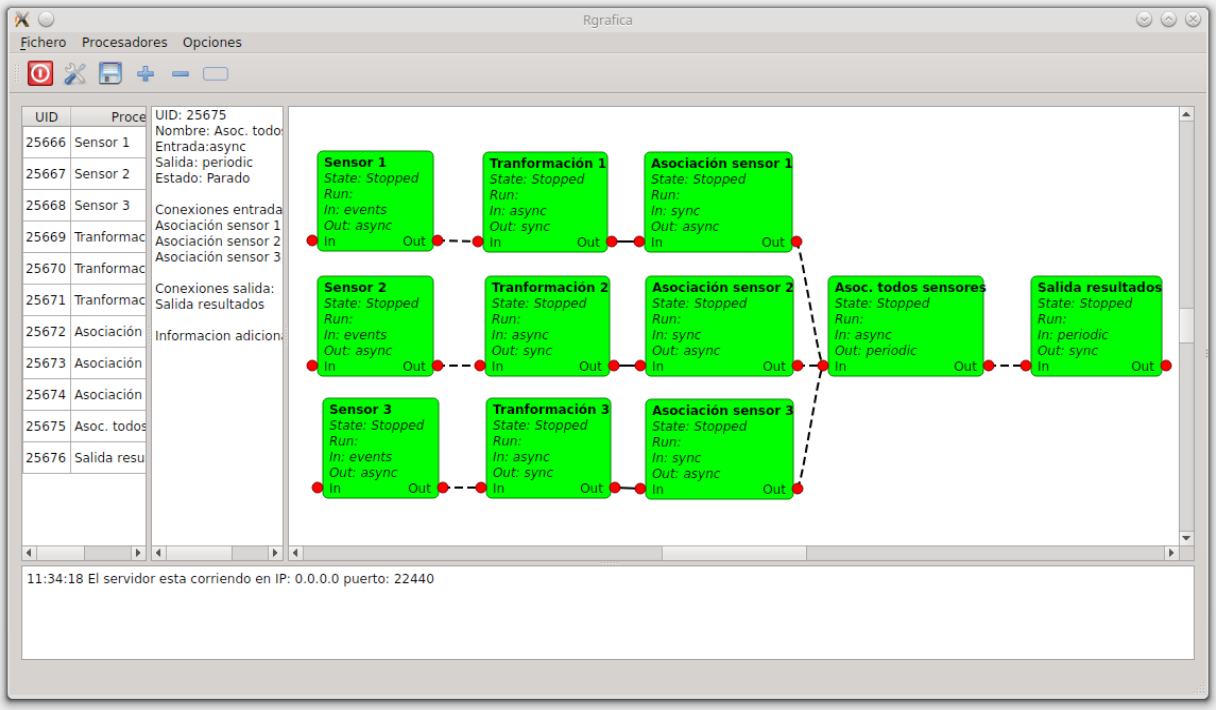

Ilustración 8: Ventana principal

Como se puede ver en la ilustración 8, que corresponde a una captura de la aplicación ejecutándose en el entorno KDE sobre Linux, en la parte superior hay una barra de menús y una barra de botones. Con ambas se pueden realizar las mismas operaciones. Están reiteradas para que en entornos como OS X o Unity sobre Linux, en los que la barra de menús no aparece en la propia ventana de la aplicaciones, estas operaciones estén accesibles fácilmente.

La operaciones que se pueden realizar con la barra de botones y los menús, son:

- Salir de la aplicación: se pide confirmación al usuario y, si este la da, se realiza la operación

descrita en la opción de guardar y termina la aplicación.

- Configurar la aplicación: permite al usuario variar algunos parámetros del funcionamiento de la aplicación. Estos parámetros se detallan en el siguiente apartado 4.2.2 Ventana de configuración.
- Guardar: guarda los *MessageProcessor* que actualmente están visibles en la aplicación y los parámetros de configuración.

  Para evitar que se tengan que volver a recibir todos los datos de los *MessageProcessor* cada vez que se inicia la aplicación, se ofrece al usuario la posibilidad de guardar los diferentes *MessageProcessor* que están visibles así como su estado, modos de gestión de la entrada y salida y conexiones entre ellos. Con respecto al estado es de destacar que, como se explicará posteriormente, se guarda la hora de inicio de la última ejecución de un procesamiento por lo que el tiempo de ejecución se mostrará correctamente aunque se cierre y se vuelva a abrir la aplicación.

  Por otra parte, también se guardan los parámetros de configuración de la aplicación en el momento actual.

  También es destacable, como se ha dicho antes, que esta operación se realiza automáticamente cada vez que el usuario termina la aplicación.
- Añadir y eliminar *MessageProcessor:* realizar estas operaciones no debería ser necesario, pero se ofrece la opción por si se pierde la sincronización de la situación que refleja la aplicación gráfica y la que realmente tiene el *framework*.
- Vaciar la representación gráfica: este botón permite eliminar todos los *MessageProcessor* de la representación gráfica. Es útil si la aplicación desarrollada se está usando para representar el funcionamiento de un sistema que se reinicia frecuentemente, por lo que es necesario borrar todos los elementos de la representación gráfica en cada reinicio.

Por último, la barra de estado proporciona una versión corta y visible durante unos segundos de los sucesos reseñables que puedan ocurrir. Con esto se consigue que el usuario, usando el separador deslizable, pueda hacer desaparecer el registro de sucesos de la ventana principal y conseguir más espacio para la visualización de los *MessageProcessor* sin perder por ello la posibilidad de tener información sobre los posibles errores que puedan ocurrir a lo largo de la ejecución de la aplicación.

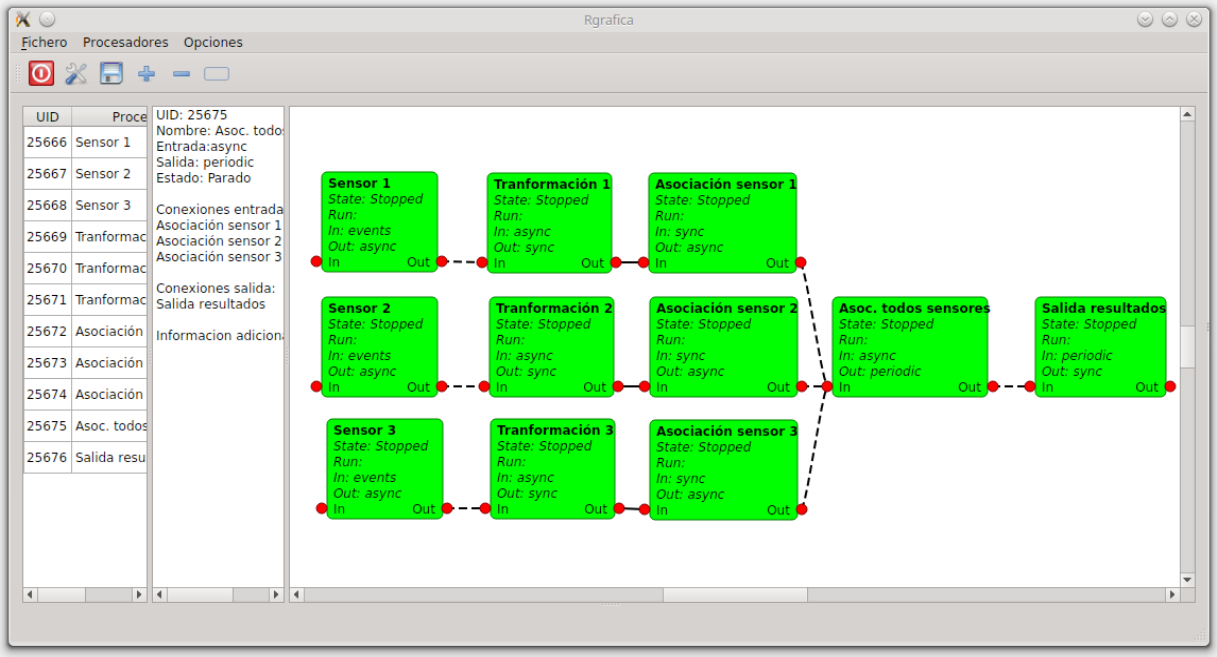

Ilustración 9: Ventana principal con el registro de sucesos oculto

En la ilustración anterior se muestra un ejemplo de como un error ocurrido mientras que el usuario no tenía el registro de sucesos visible aparece en la barra de estado. En la siguiente ilustración se observa el detalle del error que se muestra si el usuario vuelve a deslizar el separador hacia arriba para que aparezca el registro de sucesos.

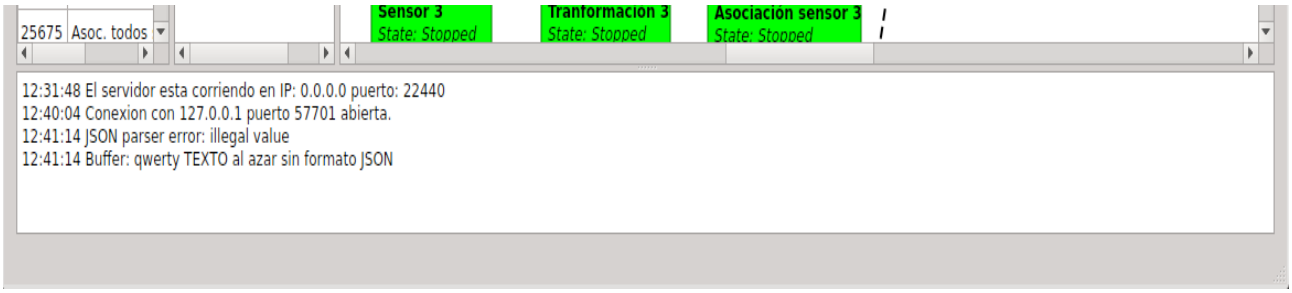

Ilustración 10: Ventana principal con detalle de error

4.2.2 Ventana de configuración

En la ventana de configuración, que se muestra en la ilustración 11, se pueden variar los siguientes parámetros de la aplicación:

- IP en la que escucha la aplicación: la aplicación desarrollada se encarga de detectar las

interfaces de red que tengan una IPv4 y permite con esta opción elegir en cual se quedará escuchando a la espera de las conexiones provenientes del *framework*. Existe la opción de que la aplicación permanezca escuchando en todas las interfaces de red.

- Puerto en el que escucha la aplicación: al igual que con la IP, en este cuadro de texto se puede elegir el puerto en el que la aplicación permanecerá escuchando a las conexiones provenientes del *framework*.
- Tamaño del registro de sucesos: con esta opción se puede elegir cuantas lineas de registro almacena el cuadro de texto del registro de sucesos de la ventana principal. Si en esta opción se elige el valor 0 el cuadro de texto se vaciará.

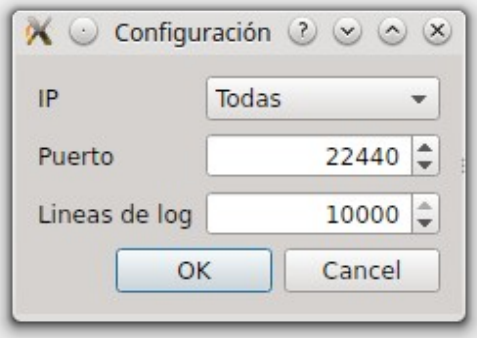

Ilustración 11: Ventana de configuración

Los parámetros modificados quedarán guardados al cerrar la aplicación o al presionar el botón para guardar y serán leídos en la siguiente ejecución de la aplicación. Si se varían las opciones de la IP o el puerto donde escucha la aplicación es necesario reiniciarla para que tomen efecto los nuevos parámetros.

4.2.3 Ventana para añadir *MessageProcessor*

Esta ventana permite al usuario añadir nuevos *MessageProcessor* a la representación gráfica sin que estos sean el resultado de una orden recibida desde el *framework*. Aunque esto no debería ser necesario nunca se ha añadido esta posibilidad para facilitar la resolución manual de las posibles perdidas de sincronización entre el estado del *framework* y la aplicación gráfica.

Como se puede observar en la ilustración 12, la ventana permite seleccionar los diferentes atributos para el *MessageProcessor* que se va a crear.

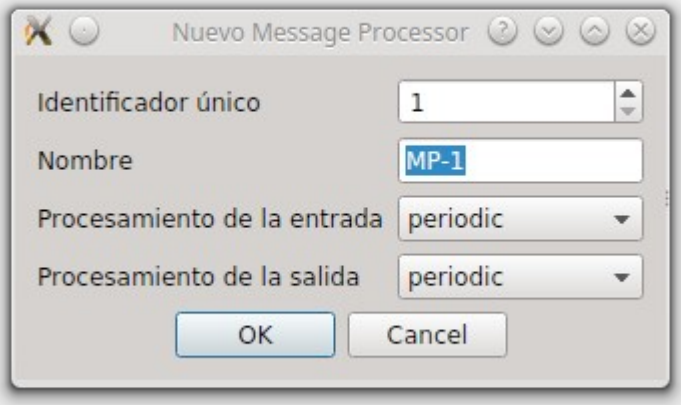

Ilustración 12: Ventana de creación de MessageProcessor

Estos atributos son:

- Identificador único: es un número entero positivo que identifica unívocamente al *MessageProcessor* que se va a crear. Por defecto la aplicación proporcionar el entero más bajo disponible. En el caso de que se elija uno existente la aplicación avisa de su existencia.

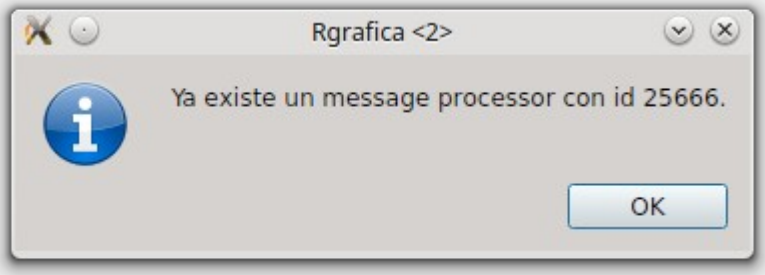

Ilustración 13: Aviso de Identificador de MessageProcessor repetido

- Nombre: es un nombre descriptivo del *MessageProcessor* que se va a crear. Pueden existir varios *MessageProcessor* con el mismo nombre, por ejemplo puede haber varios *MessageProcessor* con el nombre "Radar". Por defecto la aplicación asigna al nuevo *MessageProcessor* el nombre "MP-*id*", siendo *id* el identificador unívoco del nuevo *MessageProcessor*.
- Procesamiento de entrada: es el tipo de gestión de los datos de entrada que va a tener el

nuevo *MessageProcessor*. En el desplegable se puede elegir entre las opciones *periodic*, *events*, *sync* y *async* que corresponden respectivamente a la gestión de datos de entrada por tiempo, por eventos, síncrona y asíncrona. El uso de estos nombres para las opciones, *events* y no eventos, se debe a que son, como se detallará más adelante, las opciones que aparecen en las órdenes que se reciben del *framework*. Es decir, al usar esta ventana se está simulando el recibir una orden para la creación de un nuevo *MessageProcessor*.

- Procesamiento de salida: es el tipo de gestión de los datos de salida que va a tener el nuevo *MessageProcessor* y, al igual que en la gestión de datos de entrada, en el desplegable se puede elegir entre las opciones *periodic*, *events*, *sync* y *async*.

Si se ha creado en la interfaz gráfica un nuevo *MessageProcessor* con seguridad será necesario crear las conexiones del nuevo *MessageProcessor* con alguno de los existentes. Para hacer esto basta con pinchar en el punto que representa el puerto de entrada de un *MessageProcessor* y arrastrar el ratón hasta el punto que representa el puerto de salida de otro *MessageProcessor*. También se puede realizar a la inversa, pinchar en el puerto de salida de un *MessageProcessor* y arrastrar hasta la entrada de otro.

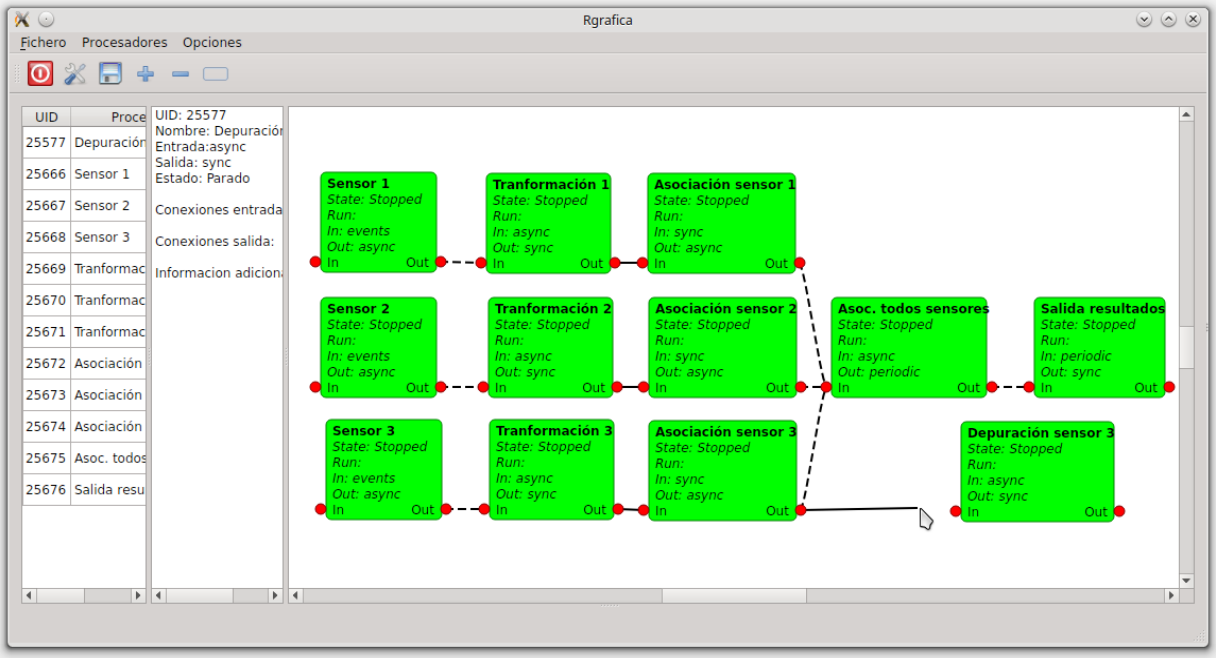

Ilustración 14: Creación de un enlace manualmente

Tal y como si hubiese llegado la orden de creación de enlace entre dos *MessageProcessor* la interfaz creará un enlace y lo dibujará con una linea continua si tanto la gestión de salida como la de entrada es síncrona o con una linea discontinua en cualquier otro caso.

Para eliminar conexiones o *MessageProcessor* de la representación gráfica basta con pincharlos con

el botón izquierdo del ratón. Los *MessageProcessor* también se pueden eliminar pulsando el botón para eliminar *MessageProcessor*, aunque en este caso se eliminará el *MessageProcessor* que esté seleccionado en la representación gráfica.

4.3 Comunicación con el *framework*

Para la comunicación con el *framework* se utiliza TCP como protocolo de transporte y JSON como protocolo de aplicación. Al ser un requisito fundamental de la aplicación que fuese multiplataforma, la elección de los protocolos a usar también tenía este requisito, el *framework* se debe poder comunicar con la aplicación gráfica indistintamente del sistema operativo en el que esta se esté ejecutando. A la hora de elegir JSON como protocolo a nivel de aplicación también se tuvo en cuenta su simplicidad y la flexibilidad que proporcionaba de cara a futuras modificaciones del protocolo.

Con respecto a TCP la aplicación desarrollada permanece a la escucha, en estado *LISTEN*, y es el *framework* el que ejerce de par activo en la conexión, es decir es el que abre y cierra las conexiones. Dicho de otra forma, la aplicación desarrollada es el servidor y el *framework* es el cliente.

La aplicación desarrollada no tiene por sí misma ningún tipo de limitación en cuanto al numero de conexiones entrantes que puede mantener simultáneamente, lo que da la posibilidad de implementar el otro extremo de la comunicación como sea más conveniente. Se puede implementar de forma que sea cada *MessageProcessor* el que envíe sus conexiones y cambios de estado o se puede implementar realizando una sola conexión desde el *framework* a la aplicación de monitorización.

La aplicación mantiene una lista de conexiones establecidas en la que cada conexión tiene una porción propia de memoria asignada. Cada vez que el sistema operativo informa de que hay datos para ser leídos en una conexión, la aplicación copia estos datos a la memoria asignada a la conexión y pasa al analizador sintáctico JSON un puntero a dicha zona de memoria. Si el analizador sintáctico encuentra un objeto JSON completo y sin errores realiza la orden contenida en él y lo elimina de la memoria asignada a la conexión. Si encuentra un objeto JSON no terminado simplemente lo deja en la memoria asignada a la conexión a la espera de que llegue la parte que falta. Y por último, si encuentra un error sintáctico o un carácter que no pertenece a un objeto JSON elimina de la memoria asignada a la conexión lo leído hasta ese carácter y muestra el error en el registro de sucesos.

Con este comportamiento se consiguen evitar falsos errores causados por la fragmentación que puede provocar la transmisión de mensajes mediante TCP y a la vez mantener informado al usuario de los posibles errores de formato que se puedan producir en la comunicación entre las aplicaciones.

Para conseguir que la implementación del código encargado de las comunicaciones fuese multiplataforma se ha utilizado la API que Qt provee para ello y que se describe en el apartado 4.4.1.2.

Además, gracias al uso del mecanismo de señales para la comunicación entre objetos propio de Qt, se ha podido desarrollar la aplicación sin múltiples hilos de ejecución y sin que la interfaz gráfica se quedara congelada mientras se realizan operaciones de red.

4.3.1 JSON

Como se ha dicho, el protocolo usado a nivel de aplicación es JSON. Este protocolo es un estándar para el intercambio de información y tiene un uso ampliamente extendido en el ámbito de las aplicaciones web debido a que es un subconjunto de la notación literal de objetos de JavaScript y que, frente a XML, es más liviano en la transmisión. Estas ventajas han propiciado su uso por empresas de servicios web con un gran número de visitas, como Google o Yahoo, lo que a su vez ha conducido a que sea usado más allá del ámbito de las aplicaciones web. En la actualidad existen analizadores sintácticos de JSON para multitud de lenguajes de programación no relacionados, en principio, con el desarrollo de aplicaciones web. En el caso que nos ocupa, Qt incorpora un analizador sintáctico JSON desde la versión 5[3].

La especificación de JSON es muy sencilla. Se presenta a continuación una versión simplificada de la RFC 7159 [5], en la que se detalla toda la especificación de JSON, para explicar posteriormente el uso que se hace en el protocolo de comunicación.

La sintaxis de JSON está formada por cuatro estructuras básicas:

- Objeto: es una lista no ordenada de cero o más parejas nombre – valor, donde:
  - Nombre es una cadena de caracteres .
  - Valor puede ser una cadena de caracteres, un número, un objeto, una lista o los valores *true*, *false* o *null*.
- Lista: es una secuencia ordenada de cero o más valores.
- Cadena de caracteres: es una secuencia de caracteres *unicode*.
- Número: es un número como está especificado en cualquier lenguaje de programación, en concreto en JavaScript, sin ceros precedentes.

Como se puede ver la especificación es muy sencilla, lo que sin duda ha ayudado a que su uso esté tan extendido. La formulación de los anteriores cuatro puntos en EBNF es:

```
objecto ::= '{' ((cadena-caracteres ':' valor ) ( ',' cadena-caracteres ':' valor )*)? '}'

lista   ::= '[' ( valor (',' valor )*)? ']'

valor   ::= ( 'true' | 'false' | 'null' | objecto | lista | número | cadena-caracteres )

cadena-caracteres ::= '"' ( caracter-UNICODE-no-escapado | '\' ( '"' | '\' | '/' | 'b' | 'f' | 'n' |
'r' | 't' | 'uXXXX' ) ) '"'
```

3 Las clases que implementan el analizador sintáctico JSON en Qt han ido cambiando rápidamente en las primeras versiones de Qt 5. La aplicación desarrollada sólo se puede compilar con la versión de Qt 5.3 o superior.

```
número ::= '-'? ( '0' | dígito1-9 dígito* ) ('.' dígito+ )? ( ('e' | 'E' ) ( '+' | '-')? dígito+ )?
```

y su representación en los respectivos diagramas de sintaxis:

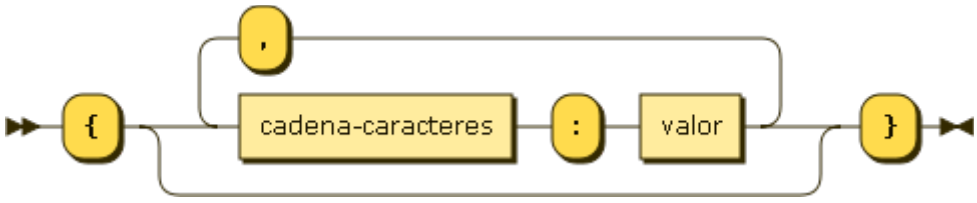

Ilustración 15: Diagrama sintáctico de un objeto JSON

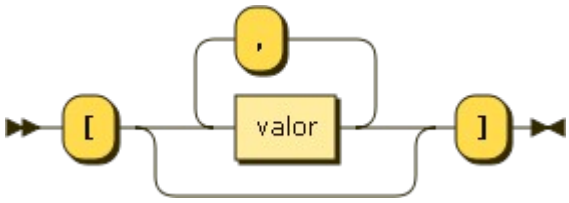

Ilustración 16: Diagrama sintáctico de una lista JSON

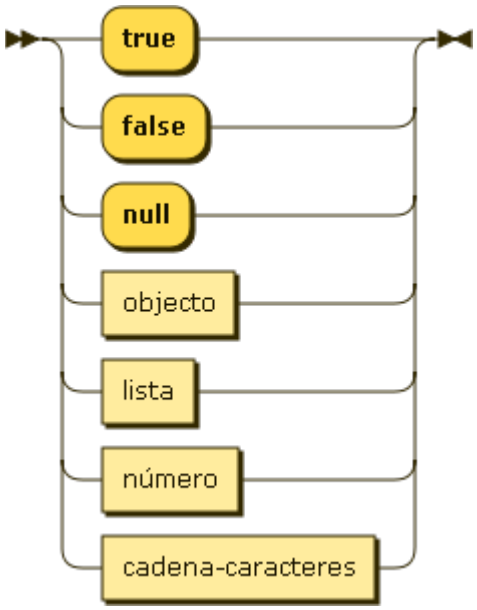

Ilustración 17: Diagrama sintáctico de un valor JSON

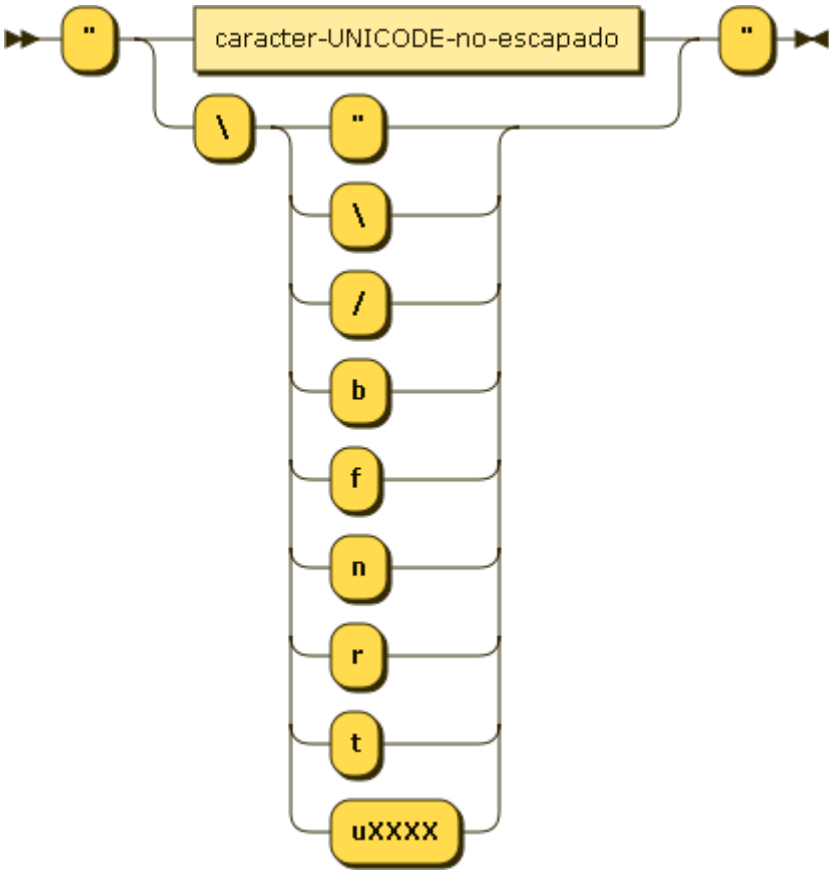

Ilustración 18: Diagrama sintáctico de una cadena de caracteres JSON

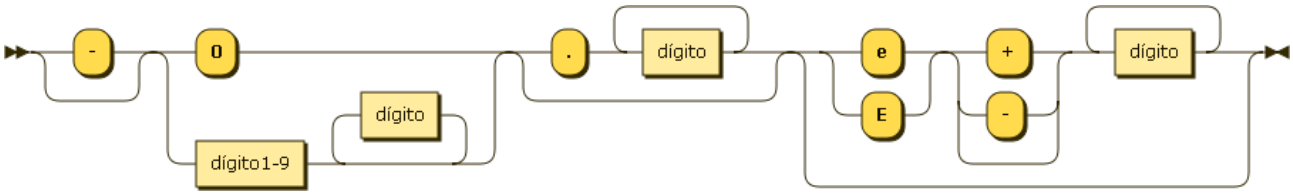

Ilustración 19: Diagrama sintáctico de un número JSON

Unos ejemplos sencillos de objetos JSON serían:

- Un objeto:

```
{
        "NombreFruta":"Manzana" ,
        "Cantidad":20
}
```

- Una lista de objetos:

```
{
"Frutas": [
          { "NombreFruta":"Manzana" , "cantidad":10 },
```

```
           { "NombreFruta":"Pera" , "cantidad":20 },
           { "NombreFruta":"Naranja" , "cantidad":30 }
          ]
}
```

- Un objeto que es una lista de listas:

```
{
"Fruteria":
           [
            {"Fruta":
             [
              {"Nombre":"Manzana","Cantidad":10},
              {"Nombre":"Pera","Cantidad":20},
              {"Nombre":"Naranja","Cantidad":30}
             ]
            },
            {"Verdura":
             [
              {"Nombre":"Lechuga","Cantidad":80},
              {"Nombre":"Tomate","Cantidad":15},
              {"Nombre":"Pepino","Cantidad":50}
             ]
            }
           ]
}
```

Como se puede observar en los ejemplos, los objetos JSON pueden ser leídos fácilmente por un humano si están correctamente sangrados.

4.3.2 El protocolo de comunicación

Usando la sintaxis de JSON, la aplicación desarrollada puede recibir las siguientes órdenes:

- Sobre un *MessageProcessor*:
  - Creación: la aplicación muestra un nuevo *MessageProcessor* en la representación gráfica. Esta operación debe incluir como parámetros el identificador único del *MessageProcessor*, su nombre, el modo de gestión de entrada de datos y el modo de gestión de salida de datos. La aplicación comprueba que no exista un *MessageProcessor* con ese identificador y muestra un error si es así. También comprueba si existe un *MessageProcessor* con ese nombre pero, en este caso, si existe sólo muestra un mensaje en el registro de sucesos y crea el *MessageProcessor*.

    Para indicar los diferentes modos de gestión de los datos de entrada o de salida la orden debe especificar una las opciones *periodic*, *events*, *sync* o *async*, que respectivamente indican un modo de gestión por tiempo, por eventos, síncrona o asíncrona.
  - Borrado: la aplicación elimina el *MessageProcessor* indicado en la orden y sus conexiones. Esta operación debe incluir únicamente el identificador del *MessageProcessor* a borrar, el resto de las parejas de valores del objeto JSON son ignorados.

La aplicación comprueba que exista un *MessageProcessor* con ese identificador y si no es así muestra un error en el registro de sucesos.

- Actualización: con esta orden la aplicación actualiza una o varias propiedades de un *MessageProcessor*. La orden debe especificar obligatoriamente el identificador del *MessageProcessor* a actualizar y opcionalmente uno o varios de los siguientes campos: el nombre, el modo de gestión de entrada de datos o el modo de gestión de salida de datos.

- Sobre una conexión entre dos *MessageProcessor*:
  - Creación: la aplicación crea una conexión desde el *MessageProcessor* indicado en el campo "desde", *from*, hasta el *MessageProcessor* indicado en el campo "hacia", *to*. Ambos campos deben ser el identificador de un *MessageProcessor* que exista pero que no tengan una conexión entre ellos. En caso contrario la aplicación muestra el error pertinente en el registro de sucesos.

    Al crear la conexión entre los *MessageProcessor* la aplicación comprueba si el modo de gestión de datos de ambos extremos es síncrono y en este caso dibuja una linea continua. En caso contrario la aplicación dibuja un linea discontinua.

  - Borrado: esta orden elimina un enlace entre los *MessageProcessor* indicados. La aplicación comprueba que la conexión existe y la elimina. En caso de que no exista uno de los *MessageProcessor* indicados o que no exista la conexión la aplicación muestra en el registro de sucesos un mensaje de error.

- Sobre el estado de un *MessageProcessor*:
  - Actualización: cada *MessageProcessor* tiene un estado de la entrada, de la salida y del procesamiento. Estos estados indican si el *MessageProcessor* está recibiendo datos, enviando datos o procesando los datos. Con esta operación se puede modificar cada uno de esos estados del *MessageProcessor*, lo que se muestra de distintas formas en la representación gráfica.

    En el caso del estado de la entrada y salida de los datos, el estado se representa por el color del puerto de entrada o salida. En caso de que el *MessageProcessor* esté recibiendo o enviando el puerto adecuado se vuelve verde y parpadea indicando actividad. Si el *MessageProcessor* no esta recibiendo o enviando el puerto adecuado se vuelve rojo, indicando que no hay actividad.

    En el caso del estado del procesamiento, se visualiza en el campo *State* del propio *MessageProcessor* cual es el estado. Además, si está realizando un procesamiento se indica en el campo *Run* el tiempo transcurrido desde que se inició el procesamiento.

Cuando la aplicación recibe el mensaje de que un *MessageProcessor* ha dejado de procesar lo indica variando el estado en el campo *State* pero deja el tiempo del último procesamiento en el campo *Run*.

Esta orden debe incluir el identificador del *MessageProcessor* del que se quiere variar el estado, el campo *action* con los valores *input*, *output* o *run* y el campo *state* con los valores *start* o *stop*.

- Sobre la información adicional de un *MessageProcessor*:
  - Actualización: con esta orden se asigna una nueva lista de parejas de atributo y valor al *MessageProcessor*. Esta lista se muestra en la parte de datos del *MessageProcessor* seleccionado de la representación gráfica.

    La orden debe incluir el identificador del *MessageProcessor* a modificar y un campo *info* seguido de una lista de objetos JSON.

  - Borrado: esta orden elimina la lista de atributos de información adicional del *MessageProcessor* indicado.

La formulación en EBNF de los puntos anteriores es:

```
orden ::=
'{' '"type"' ':'
( '"processor"' ',' '"operation"' ':'
     ( '"create"' ',' '"id"'          ':' número ','
                      '"name"'        ':' cadena-caracteres ','
                      '"input_mode"'  ':' ('"periodic"' | '"events"' | '"sync"' | '"async"' ) ','
                      '"output_mode"' ':' ('"periodic"' | '"events"' | '"sync"' | '"async"' ) |
       '"delete"' ',' '"id"' ':' número |
       '"update"' ',' '"id"' ':' número ','
           ( ( '"name"'        ':' cadena-caracteres |
               '"input_mode"'  ':' ('"periodic"' | '"events"' | '"sync"' | '"async"' ) |
               '"output_mode"' ':' ('"periodic"' | '"events"' | '"sync"' | '"async"' ))
                ( ',' ( '"name"'        ':' cadena-caracteres |
                     '"input_mode"'  ':' ('"periodic"' | '"events"' | '"sync"' | '"async"' ) |
                     '"output_mode"' ':' ('"periodic"' | '"events"' | '"sync"' | '"async"' ) )
             )*
           )
     ) |
  '"link"'            ',' '"operation"' ':' ( '"create"' | '"delete"' ) '"from"' ':' número ','
                                                                          '"to"'   ':' número     |
  '"processor_state"' ',' '"operation"' ':' '"update"' ','
                          '"action"'    ':' ( '"input"' | '"output"' | '"run"' ) ','
                          '"state"'     ':' ( '"start"' | '"stop"' )                            |
  '"processor_info"'  ',' '"operation"' ':' ( '"update"' '"id"' ':' número ',' objeto |
                                              '"delete"' '"id"' ':' número )
)
' }'
```

Se muestra una versión simplificada, puesto que el analizador sintáctico permite que los campos de una misma operación no estén en un orden en concreto, pueden variar su posición siempre que respeten la sintaxis JSON.

El diagrama de sintaxis del protocolo de comunicaciones se muestra en la figura 20.

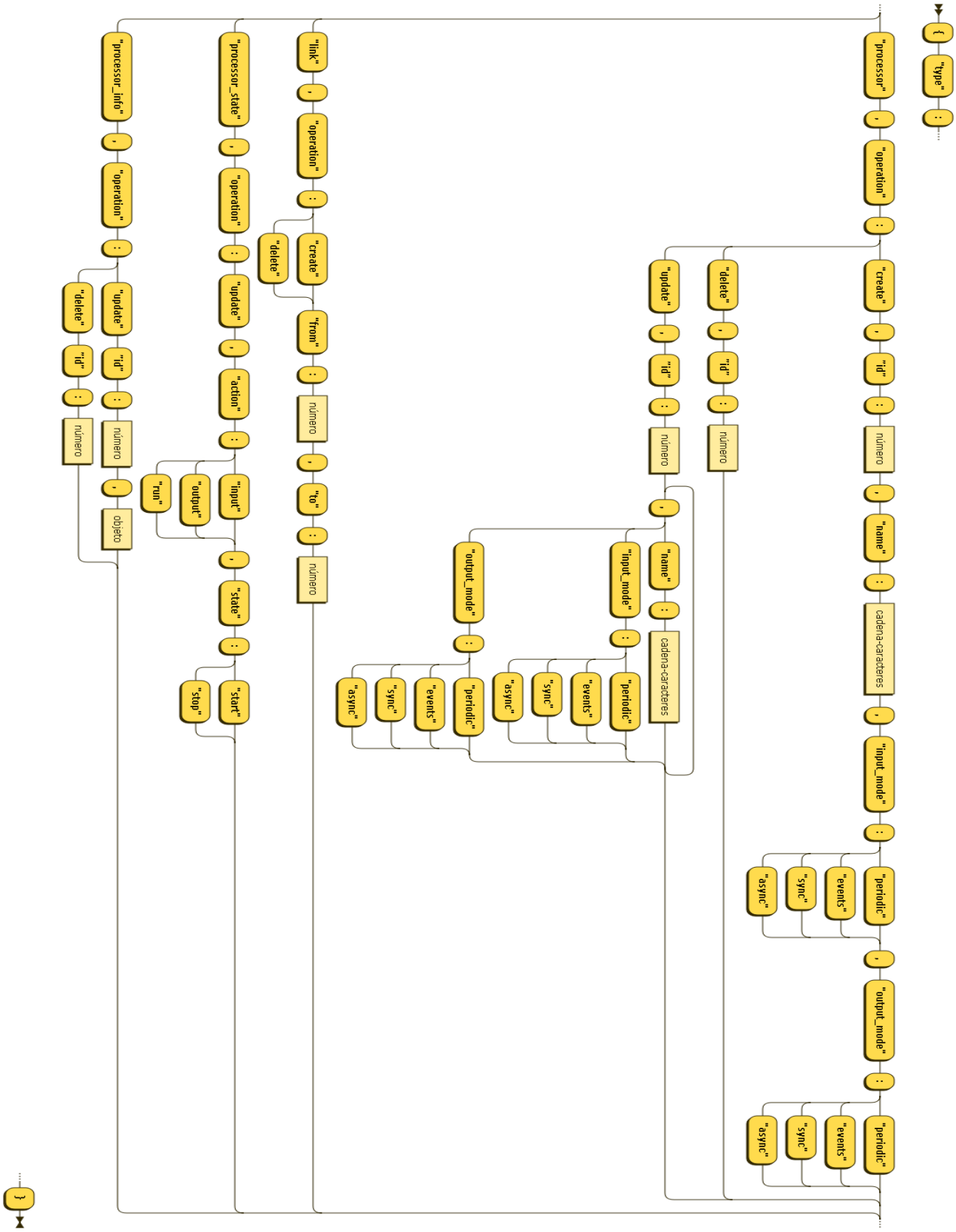

Ilustración 20: Diagrama sintáctico del protocolo de comunicaciones

Como ejemplos de mensajes con la especificación descrita se muestran los mensajes necesarios para crear la representación gráfica que se puede ver en la ilustración 8.

Creación de los sensores

```
{
"operation":"create",
"type":"processor",
"id": 25666,
"name":"Sensor 1",
"input_mode":"events",
"output_mode":"async"
}
{
"operation":"create",
"type":"processor",
"id": 25667,
"name":"Sensor 2",
"input_mode":"events",
"output_mode":"async"
}
{
"operation":"create",
"type":"processor",
"id": 25668,
"name":"Sensor 3",
"input_mode":"events",
"output_mode":"async"
}
```

Creación de transformaciones cartesianas

```
{
"operation":"create",
"type":"processor",
"id": 25669,
"name":"Tranformación 1",
"input_mode":"async",
"output_mode":"sync"
}
{
"operation":"create",
"type":"processor",
"id": 25670,
"name":"Tranformación 2",
"input_mode":"async",
"output_mode":"sync"
}
{
"operation":"create",
"type":"processor",
"id": 25671,
"name":"Tranformación 3",
"input_mode":"async",
"output_mode":"sync"
}
```

Creación de asociaciones con sensores

```
{
"operation":"create",
"type":"processor",
"id": 25672,
"name":"Asociación sensor 1",
"input_mode":"sync",
"output_mode":"async"
}
{
"operation":"create",
"type":"processor",
"id": 25673,
"name":"Asociación sensor 2",
"input_mode":"sync",
"output_mode":"async"
}
{
"operation":"create",
"type":"processor",
"id": 25674,
"name":"Asociación sensor 3",
"input_mode":"sync",
"output_mode":"async"
}
```

Creación asociación todos sensores y salida

```
{
"operation":"create",
"type":"processor",
"id": 25675,
"name":"Asoc. todos sensores",
"input_mode":"async",
"output_mode":"periodic"
}
{
"operation":"create",
"type":"processor",
"id": 25676,
"name":"Salida resultados",
"input_mode":"periodic",
"output_mode":"sync"
}
```

Creación enlaces entre sensores y transformaciones cartesianas

```
{
"operation":"create",
"type":"link",
"from": 25666,
"to": 25669
}
{
"operation":"create",
"type":"link",
"from": 25667,
"to": 25670
}
{
"operation":"create",
"type":"link",
"from": 25668,
"to": 25671
}
```

Creación enlaces entre transformaciones cartesianas y asociaciones

```
{
"operation":"create",
"type":"link",
"from": 25669,
"to": 25672
}
{
"operation":"create",
"type":"link",
"from": 25670,
"to": 25673
}
{
"operation":"create",
"type":"link",
"from": 25671,
"to": 25674
}
```

Creación enlaces entre asociaciones y asociación de todos los sensores

```
{
"operation":"create",
"type":"link",
"from": 25672,
"to": 25675
}
{
"operation":"create",
"type":"link",
"from": 25673,
"to": 25675
}
{
"operation":"create",
"type":"link",
"from": 25674,
"to": 25675
}
```

Creación enlace entre asociación de todos los sensores y salida

```
{
"operation":"create",
"type":"link",
"from": 25675,
"to": 25676
}
```

4.4 Diseño del sistema de monitorización

Como se mencionado anteriormente, la aplicación desarrollada sólo tiene un hilo de ejecución a pesar de que atiende simultáneamente a la comunicación por red y a la interfaz gráfica de usuario. La implementación de esto ha sido sumamente sencilla gracias al mecanismo de señales y ranuras[4] que ofrece Qt. El no ser una aplicación con múltiples hilos ha simplificado significativamente el desarrollo.

Este mecanismo de señales y ranuras es similar al que ofrece la librería *Boost* [6] para C++, que ha sido usada en el desarrollo del *framework* de procesamiento de información descrito en el capítulo 3, o al que ofrece C# con el nombre de eventos y delegados. Consiste en que un objeto puede exponer algunos de sus métodos, las ranuras, para que sean llamados cuando otro objeto emita un evento, la señal. Este mecanismo se usa típicamente en la programación de aplicaciones gráficas, cuando un objeto que implementa un control gráfico, como un botón, necesita llamar al método de otro objeto, por ejemplo la rutina para cerrar la ventana principal, manteniendo el nivel de encapsulamiento.

Qt hace un uso extensivo de este mecanismo y lo emplea, además de para la programación de los controles gráficos, en multitud de situaciones como para implementar la entrada y salida en los *sockets* o para implementar los eventos de un temporizador.

Siendo una aplicación con un solo hilo de ejecución y desarrollada siguiendo los estándares de Qt, sólo se instancia un objeto de la clase *Qapplication* que se encarga del hilo de ejecución y de despachar los eventos al resto de los objetos. Los eventos de interés para el caso de la aplicación desarrollada son principalmente los eventos provenientes de la interfaz gráfica y los eventos provenientes de la red.

En la ilustración 21 se muestra un diagrama de clases simplificado de la aplicación. Las clases principales son:

- *MainWindow*: sólo se instancia un objeto de esta clase. Esta clase implementa la mayoría de la lógica de la aplicación puesto que en ella se implementan las ranuras para los eventos de los controles gráficos, las ranuras para los eventos de red y los métodos para analizar los mensajes que se reciben por red. En base a los mensajes que llegan por la red se crean, eliminan o modifican los elementos pertinentes en la representación gráfica.

  El constructor de esta clase también se encarga de instanciar un objeto de la clase *Ui::Mainwindow* que muestra los controles gráficos de la ventana principal. El código de estas clases es generado automáticamente por la cadena de herramientas de compilación de Qt.

- *QNodesEditor*, *QNEBlock*, *QNEPort* y *QNEConnection*: estas clases implementan cada uno

4 Se ha optado por traducir el nombre en inglés *signals and slots mechanism*, que es como normalmente se conoce.

de los elementos que se muestran en la representación gráfica de la ventana principal. Respectivamente son: el lienzo donde muestran los elementos de la representación gráfica, los *MessageProcessor*, los puertos de entrada y salida de los *MessageProcessor* y las conexiones entre *MessageProcessor*.

- *DialogNuevoMessageProcessor*, *DialogConfiguracion*: estas clases implementan la lógica de la ventana para crear un nuevo *MessageProcessor* y de la ventana para cambiar la configuración.
- Las clases del espacio de nombres *Ui*: son las clases de los controles gráficos. Como se ha mencionado antes, estas clases son generadas automáticamente por la Suite de Compilación de Qt [7] a partir del diseño realizado con la herramienta gráfica que Qt provee para ello.

Como se puede observar en la ilustración 21, en el lado izquierdo del diagrama están las clases del espacio de nombres *Ui* que implementan los controles gráficos. Estas son las clases encargadas de la interacción entre el usuario y la interfaz gráfica Estas clases utilizan la API que Qt ofrece para la creación de interfaces gráficas, como pueden ser las clases *QDialog*, *QStatusBar*, *QMenuBar*, *QGraphicsView*... Esta última es sobre la que se apoya *QNodesEditor* para gestionar la interacción entre el usuario y los diferentes elementos de la representación gráfica.

En el centro aparece la clase *MainWindow*, que es derivada de la clase *QMainWindow* que Qt ofrece para crear la ventana principal de una aplicación con la apariencia clásica. La clase *MainWindow* usa las clases que Qt ofrece para las comunicaciones TCP, en concreto *QTcpServer* para crear el *socket* de escucha y a partir del cual se crean las nuevas conexiones con el *framework*. Estas nuevas conexiones son objetos del tipo *QTcpSocket* que se guardan en una lista de registros del tipo *TClient* en el que se asocian clientes TCP con una porción de memoria asignada para guardar los datos recibidos hasta que puedan ser analizados por el analizador sintáctico JSON.

En base a los mensajes recibidos desde el *framework* y de la interacción con el usuario a través de la interfaz gráfica, *MainWindow* crea, modifica o elimina objetos de las clases que representan los elementos gráficos que se muestran en la representación gráfica. Estos objetos son de las clases *QNEBlock*, *QNEPort* y *QNEConnection*. La representación gráfica en sí misma es un lienzo de la clase QGraphicsScene que es visible usando una instancia de la clase *QGraphicsView*. La interacción del usuario con los elementos visibles en la instancia de *QGraphicsView* se realiza a través de *QNodesEditor.*

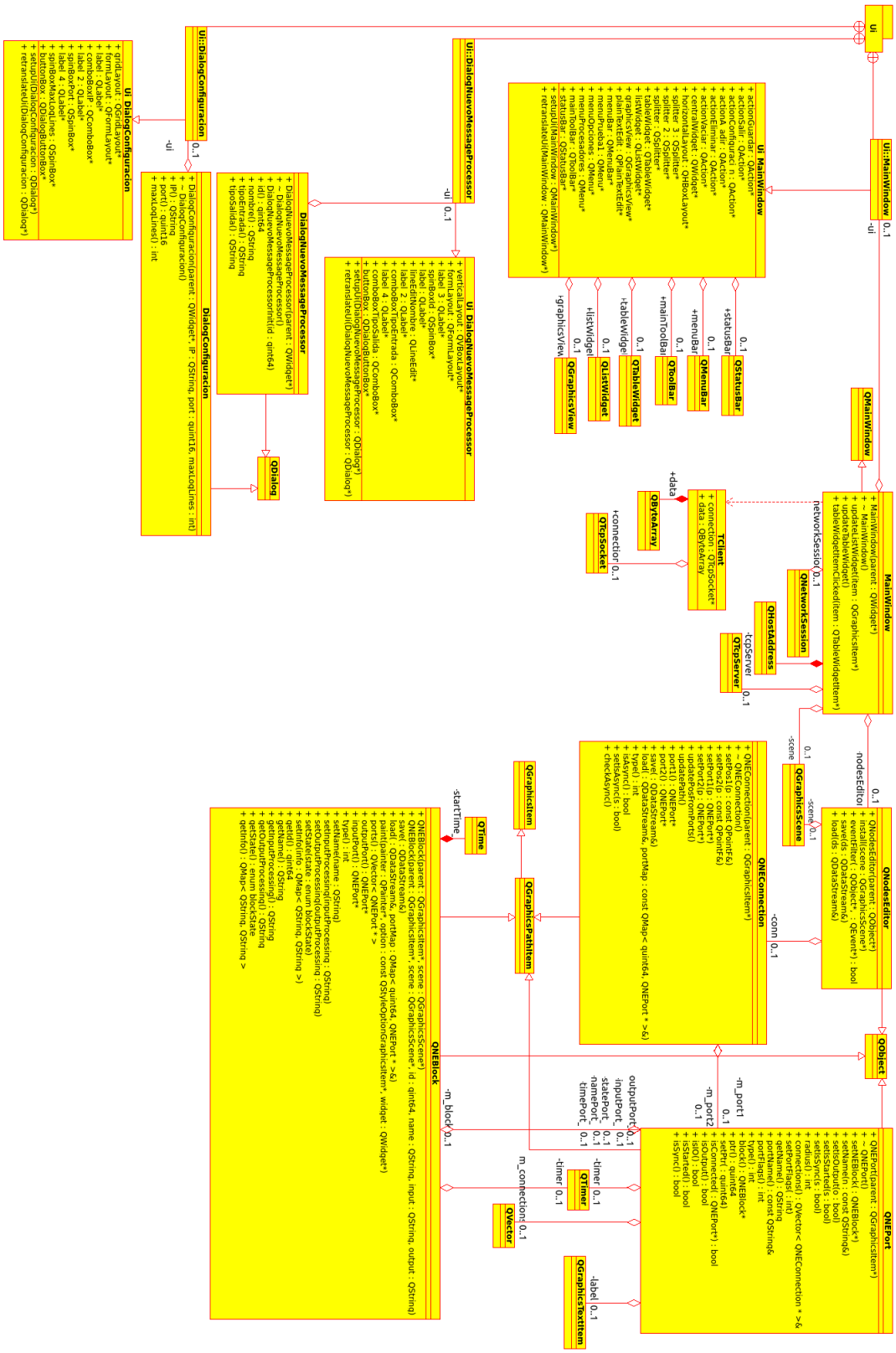

Ilustración 21: Diagrama de clases simplificado del sistema de monitorización

4.4.1 La clase *MainWindow*

En la clase *MainWindow* se implementa el núcleo del sistema de monitorización. Posee los métodos y ranuras que se encargan de dar la funcionalidad requerida, es decir recibir los mensajes provenientes del *framework* y realizar los cambios oportunos en la representación gráfica. Así mismo gestiona los eventos provenientes de los controles gráficos para proporcionar la respuesta adecuada al usuario.

La clase *MainWindow* posee una serie métodos que implementan el bloque central de la funcionalidad de la aplicación y que realizan las siguientes funciones:

- Ranuras para la gestión de los eventos de los controles gráficos.
- Ranuras para la gestión de los eventos de red.
- Métodos privados para escribir y leer la configuración de la aplicación.
- Métodos privados para el análisis de los mensajes de red y su ejecución.

Al iniciarse la ejecución del sistema de monitorización el constructor de la clase *MainWindow* se encarga de:

- Crear todos los controles gráficos que se muestran en la ventana principal usando las clases del espacio de nombres *Ui*. Estas clases son generadas automáticamente por la orden *qmake* a partir del fichero XML creado con *Qt Designer*. Tanto *qmake* como *Qt Designer* están integrados en el entorno de desarrollo *Qt Creator* que Qt provee. Desde el punto de vista del desarrollador, la creación de la interfaz gráfica en Qt consiste en colocar los elementos gráficos usando el ratón y añadir dos líneas de código al principio del constructor de la clase. De hecho, si se usa el asistente para la creación de aplicaciones de *Qt Creator* y se le indica "*Qt Widgets Application*" el código inicial de la aplicación ya incluye las lineas necesarias.
- Instanciar un objeto de la clase *QNodesEditor* e inicializarlo para que capture los eventos del control gráfico de la clase *QgraphicsScene*. Esta clase, y el resto que se usan en la representación gráfica de los *MessageProcessor*, se detallan en el apartado 4.4.2.
- Leer la configuración que ha sido escrita previamente en una ejecución anterior de la aplicación. Estos métodos se detallan en el apartado 4.4.1.1.
- Crear el *socket* para la escucha en red. Esta parte del desarrollo se explica detalladamente en el apartado 4.4.1.2.

Una vez que terminado el constructor de *MainWindow* la aplicación queda a la espera de los eventos que puedan generar nuevos mensajes llegados por la red o los eventos generados por el usuario al usar la interfaz gráfica. En base a estos eventos se ejecutan diferentes subrutinas.

En cuanto a los eventos de la interfaz gráfica, el código desarrollado se limita a implementar las funcionalidades mencionadas en la descripción de la interfaz del apartado 4.2.

En cuanto a los eventos de red, la instancia de *MainWindow* se encarga de gestionar la lista de conexiones con los clientes y de guardar en una memoria intermedia los datos que llegan por cada conexión. Hecho esto, analiza sintácticamente los mensajes llegados por red, como se ha explicado anteriormente en el apartado 4.3, y ejecuta las órdenes que estos contengan. Esta parte se detalla en el apartado 4.4.1.3.

4.4.1.1 Configuración persistente

Una de las razones por las que se ha elegido Qt sobre otras bibliotecas de controles gráficos multiplataforma ha sido el que incorpora, además de los controles gráficos, otras API que le añaden funcionalidades multiplataforma que liberan al desarrollador de escribir código dependiente de la plataforma.

Una de esas API es la ofrecida por la clase *QSettings* que permite guardar los ajustes de la aplicación independientemente de la plataforma en la que se ejecute. Por otro lado, la clase *QDataStream* ofrece una API de uso muy sencillo para serializar[5] objetos. El uso conjunto de estas dos clases permiten al desarrollador guardar de forma persistente todo tipo de valores y objetos en la configuración persistente de la aplicación.

En la clase *MainWindow* se definen los métodos *writeSettings* y *readSettings* que son encargados de escribir y leer la configuración de la aplicación.

El método *writeSettings* usa la clase *QSettings* para guardar la posición y tamaño de la ventana principal y los distintos parámetros de configuración que el usuario especifique en la ventana de configuración, es decir la IP en la que la aplicación escucha, el puerto y la longitud máxima del mensaje JSON que la aplicación puede recibir.

Además el método *writeSettings* llama al método *save* de la instancia de *QNodesEditor* de *MainWindow* para grabar la posición y el estado de los ítems que se están mostrando en la representación gráfica. El único parámetro que utiliza en esa llamada es un objeto de la clase *QDataStream* cuyo contenido posteriormente se guarda como un parámetro más de la configuración de la aplicación. La versatilidad de esta clase permite que se puedan guardar de una forma muy sencilla todos los elementos de la representación gráfica. El proceso, mostrado en la ilustración 21, es el siguiente:

1. Desde el método *writeSettings* de *MainWindow* se llama al método *save* de la instancia de *QnodesEditor* pasándole un puntero a un objeto de la clase *QDataStream*.
2. El método *save* de la instancia de *QNodesEditor* llama a los métodos *save* de todos los

5 Se ha optado por el neologismo *serializar* para traducir el término inglés *marshalling*.

MessageProcessor y conexiones de la escena pasándoles un puntero al objeto de la clase *QdataStream* que ha recibido de *MainWindow*.

3. El método *save* de cada *MessageProcessor*, que están implementados en la clase *QNEBlock*, se encarga de escribir su estado y el de todos sus puertos en la memoria propia del objeto del tipo *QdataStream*.
4. El método *save* de cada conexión, que están implementadas en la clase *QNEConnection,* escribe los puertos que forman cada conexión y su estado en la memoria propia del objeto del tipo *QdataStream*.
5. El método *writeSettings* de *MainWindow* escribe en la configuración persistente el contenido del objeto de la clase *QdataStream* en el que todos los objetos de la representación gráfica han escrito su estado y su posición en la escena.

Como se puede comprobar, el que la clase *QDataStream* sea tan flexible como para permitir *serializar* cualquier clase indistintamente de sus atributos permite guardar toda la representación gráfica de una forma muy concisa y sencilla.

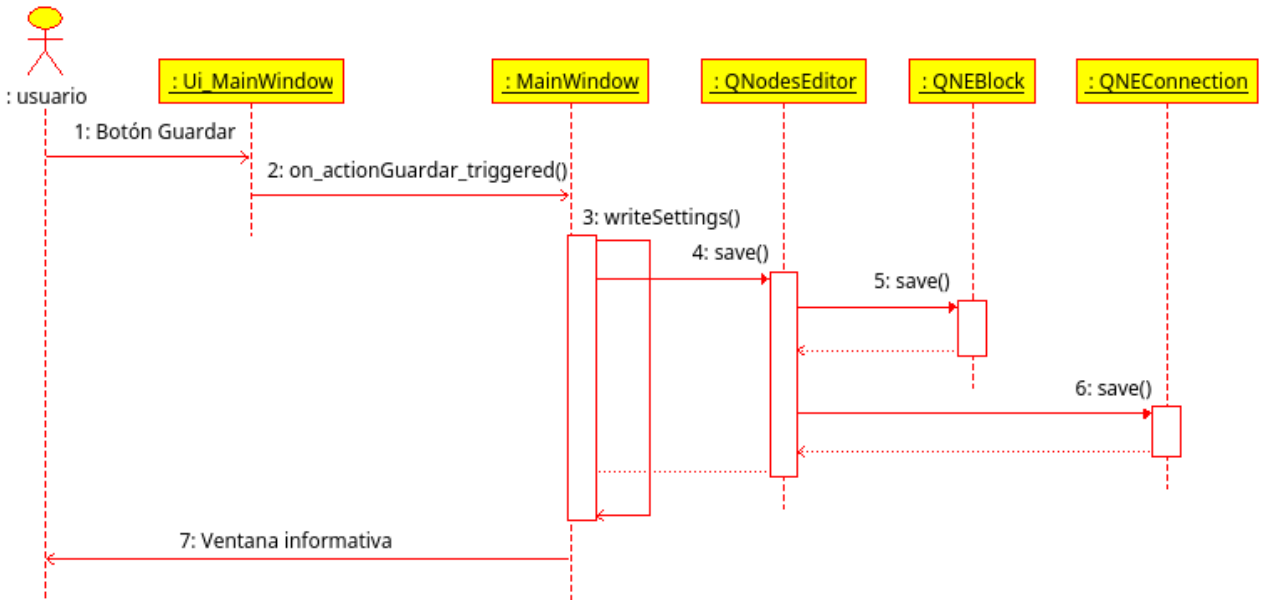

Ilustración 22: Diagrama de secuencia del guardado de la configuración

La lectura de la configuración se realiza de forma análoga a la escritura. El método *readSettings* de *MainWindow* lee todos los parámetros guardados en la configuración persistente usando la clase *QSettings* y los asigna a los atributos correspondientes para que tomen efecto. La lectura de los estados y posiciones de los elementos de la representación gráfica se realiza con un proceso similar a la escritura pero reemplazando las operaciones de escritura por operaciones de lectura realizadas

sobre un objeto de la clase *QdataStream* cuyo contenido ha sido leído de la configuración persistente.

La localización física de la configuración no aparece en el código de la aplicación. Es Qt quien se encarga de guardar los parámetros en el sitio adecuado dependiendo del sistema operativo en el que se esté ejecutando la aplicación. Para los diferentes sistemas operativos en los que se ha probado la aplicación estas localizaciones son:

- en Linux la configuración se guarda en el archivo $HOME/.config/Rgrafica/Rgrafica.conf. El contenido se guarda en texto plano con el formato:

```
[General]
maxloglines=10000
message_processors=@ByteArray()
pos=@Point(200 200)
size=@Size(400 400)
splitter1=@ByteArray(\0\0\0\xff\0\0\0\x1\0\0\0\x2\0\0\x1\0\0\0\x1\0\x1\0\0\0\x4\
x1\0\0\0\x1\0)
splitter2=@ByteArray(\0\0\0\xff\0\0\0\x1\0\0\0\x2\0\0\x2\0\0\0\x3\0\x1\0\0\0\x4\
x1\0\0\0\x1\0)
splitter3=@ByteArray(\0\0\0\xff\0\0\0\x1\0\0\0\x2\0\0\x2@\0\0\0\xc0\x1\0\0\0\x4\
x1\0\0\0\x2\0)

[Network]
IP=0.0.0.0
jsonMaxSize=1048576
port=22440
```

- en OS X la configuración se guarda en el directorio $HOME/Library/Preferences/ en el archivo com.rgrafica.Rgrafica.plist. El contenido del archivo tiene un formato nativo de OS X llamado *plist* y pude ser mostrado usando la orden *plutil* desde una terminal:

```
$ plutil -p com.rgrafica.Rgrafica.plist
{
  "size" => "@Size(915 616)"
  "maxloglines" => 10000
  "pos" => "@Point(1 68)"
  "message_processors" => <>
  "splitter1" => <000000ff 00000001 00000002 0000004f 0000004f 07010000 000100>
  "splitter2" => <000000ff 00000001 00000002 000000a5 00000340 07010000 000100>
  "splitter3" => <000000ff 00000001 00000002 00000235 0000006d 07010000 000200>
  "Network.IP" => "0.0.0.0"
  "Network.port" => 22440
  "Network.jsonMaxSize" => 1048576
}
```

- en MS-Windows en las siguientes rutas del registro:

```
HKEY_CURRENT_USER\Software\Rgrafica\Rgrafica\maxloglines
HKEY_CURRENT_USER\Software\Rgrafica\Rgrafica\pos
HKEY_CURRENT_USER\Software\Rgrafica\Rgrafica\size
HKEY_CURRENT_USER\Software\Rgrafica\Rgrafica\message_processors
HKEY_CURRENT_USER\Software\Rgrafica\Rgrafica\splitter1
```

```
HKEY_CURRENT_USER\Software\Rgrafica\Rgrafica\splitter2
HKEY_CURRENT_USER\Software\Rgrafica\Rgrafica\splitter3
HKEY_CURRENT_USER\Software\Rgrafica\Rgrafica\Network\IP
HKEY_CURRENT_USER\Software\Rgrafica\Rgrafica\Network\port
HKEY_CURRENT_USER\Software\Rgrafica\Rgrafica\Network\jsonMaxSize
```

Como se puede observar, mientras que el código no varía en absoluto, tanto el formato como la localización de la configuración de la aplicación varían considerablemente dependiendo de la plataforma en la que se ejecuta la aplicación. Además, las rutas y formatos son los propios de cada sistema operativo.

4.4.1.2 Comunicación por red

Todo el código relativo a las comunicaciones a nivel TCP se ha desarrollado usando la API que Qt proporciona para ello. Se han evitado de esta forma todos los problemas que hubieran surgido durante el desarrollo al tener que escribir código de red multiplataforma. Además, al usar las propias clases de Qt, el código desarrollado se integra perfectamente con el resto de la aplicación.

La clases de Qt usadas para la implementación del código de red han sido principalmente *QTcpServer* y *QTcpSocket*. Estas clase permiten la comunicación mediante TCP de forma no bloqueante, lo que permite que la interfaz gráfica no se congele mientras el bucle principal de la aplicación está realizando alguna operación de red, y usan el mecanismo de señales y ranuras para comunicar los cambios de estado y los errores de la capa de red.

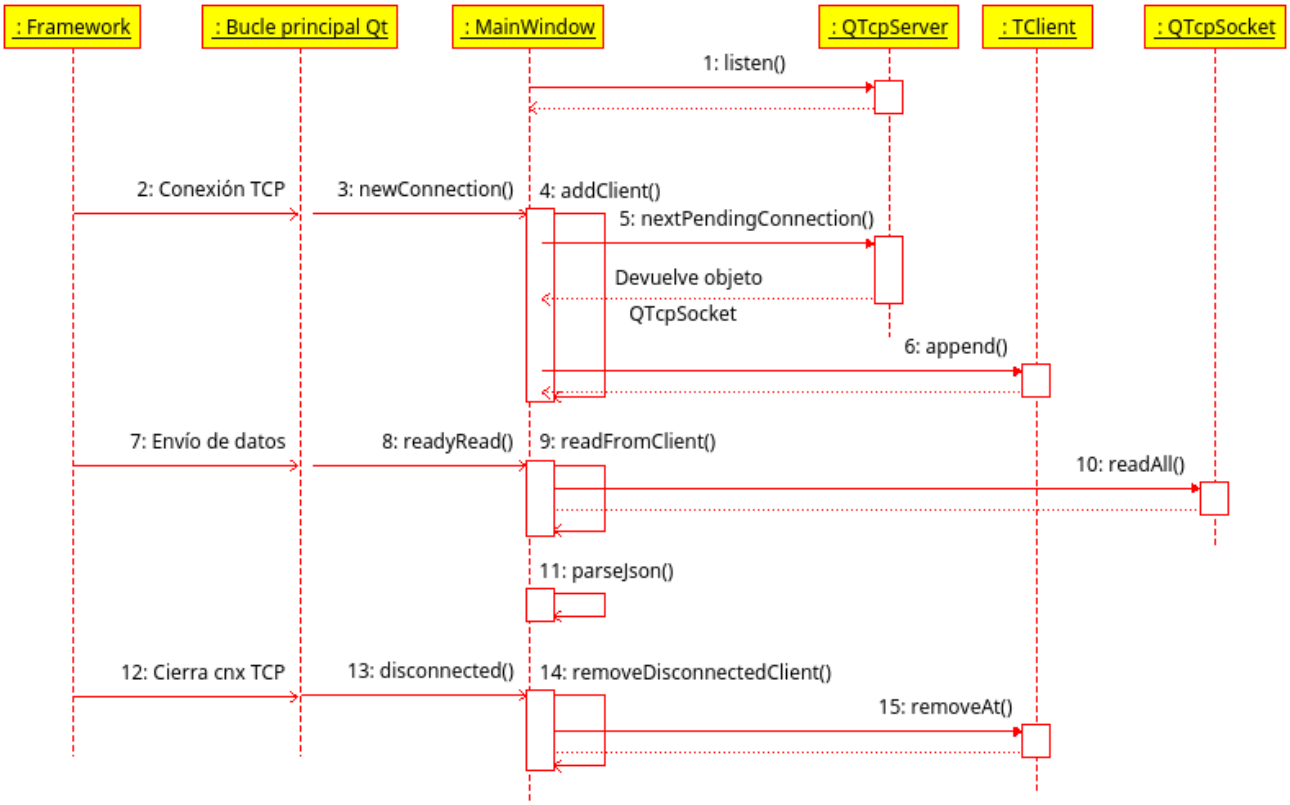

Ilustración 23: Diagrama de secuencia de una conexión TCP

Como primer paso para poder comunicarse con el *framework* el constructor de *MainWindow* crea un objeto de la clase *QTcpServer* y llama a su método *listen*. Este método es equivalente a la función *listen* [8] de los *sockets* POSIX, es decir hace que el *socket* sea pasivo o, lo que es lo mismo, pone el *socket* a escuchar conexiones entrantes. Cada vez que una de estas conexiones entrantes esté disponible el objeto de la clase *QTcpServer* enviará la señal *newConnection* que provocará que se ejecute la ranura *addClient* de *MainWindow*.

Como parámetros de la llamada *listen* figuran la IP y el puerto donde la aplicación va a escuchar las conexiones entrantes. Estos parámetros son parte de la configuración de la aplicación que son leídos del almacenamiento persistente, como se ha detallado en 4.4.1.1, y que pueden ser modificados por el usuario a través de la interfaz gráfica.

Como se ha dicho, el constructor de *MainWindow* conecta la señal que emite el objeto de la clase *QTcpServer* cuando hay disponible una nueva conexión a la ranura *addClient* de *MainWindow*. Esta ranura recoge la nueva conexión entrante, en forma de un objeto de la clase *QTcpSocket*, llamando al método *nextPendingConnection* de *QtcpServer* y lo añade al final de la lista de clientes. El método *nextPendingConnection* es análogo a la llamada *accept* [9] de los *sockets* POSIX, es decir devuelve la primera conexión que esté pendiente en la cola de conexiones.

La lista de clientes está formada por una serie de registros que contienen cada uno un puntero a un objeto de la clase *QTcpSocket*, que es una conexión con un cliente, y una porción de memoria que se destinará a almacenar los datos que se vayan recibiendo por esa conexión.

Además de guardar la nueva conexión en la lista de clientes, la ranura *addClient* conecta varias señales del objeto de la nueva conexión con diversas ranuras:

- La señal *readyRead* de la nueva instancia de *QTcpSocket* es conectada con la ranura *readFromClient* de *MainWindow*. Esta señal es emitida por *QTcpSocket* cada vez que hay nuevos datos disponibles para ser leídos en la conexión. La ranura a la que se conecta se encarga de leer esos datos, añadirlos a la porción de memoria que tiene asignada la conexión en la lista de clientes y pasar esa porción de memoria al analizador sintáctico JSON que se detalla en 4.4.1.3.
- La señal *error* de la nueva instancia de *QTcpSocket* es conectada con la ranura *removeClient* de *MainWindow*. Esta señal es emitida por *QTcpSocket* cada vez que ocurre un error de red en la conexión, como por ejemplo que el cliente cierre la conexión o que se sobrepase el tiempo de espera de la conexión. La ranura se encarga de leer todos los datos que pudieran quedar por leer, añadirlos a la porción de memoria que tiene asignada la conexión en la lista de clientes y pasar esa porción de memoria al analizador sintáctico JSON. Si el analizador sintáctico no ha podido analizar todos los datos enviados por el cliente se muestra un error en el registro de sucesos. Finalmente se elimina el cliente de la lista de clientes.

- La señal *disconnected* de la nueva instancia de *QTcpSocket* es conectada con la ranura *removeDisconnectedClient* de *MainWindow* y con la ranura *deleteLater* del mismo objeto de la conexión. La señal *disconnected* es emitida por *QTcpSocket* cuando el *socket* ha sido desconectado, sea por el motivo que sea. La ranura que captura esta señal, *removeDisconnectedClient*, únicamente comprueba si la conexión aún permanece en la lista de clientes y la borra si es así. De esta manera se previene que puedan quedar conexiones no conectadas en la lista de conexiones.

  La señal *disconnected* de la nueva instancia de *QtcpSocket* también se conecta a la ranura *deleteLater* para que el objeto sea eliminado ejecutando un *delete* cuando la aplicación alcanza el bucle principal de Qt. Esta ranura es propia de la clase *QObject* de la que es derivada la clase *QTcpSocket* e implementa un borrado seguro de objetos. De esta manera se evita que haya objetos *QTcpSocket* ocupando memoria que no están en la lista de clientes.

Como se ha detallado en este apartado, el código de gestión de los mensajes de red a nivel TCP mantiene una lista de clientes con su respectiva porción de memoria en la que se guardan los mensajes que cada cliente va recibiendo. Esta porción de memoria es la que se pasa al analizador sintáctico JSON, que se detalla en el apartado 4.4.1.3, para que las órdenes que se reciban del *framework* sean ejecutadas y, de esta forma, visualizadas en la representación gráfica.

4.4.1.3 Análisis de mensajes JSON

Una vez que la aplicación ha leído los datos provenientes del cliente, es decir del *framework*, los analiza para poder interpretar las órdenes recibidas. Aunque las conexiones utilizan el protocolo TCP, que es confiable y elimina la posibilidad de que los datos lleguen desordenados o de que haya perdida de datos, no garantiza que los objetos JSON lleguen completos en sí mismos. La aplicación tiene que encargarse de esperar a que los objetos JSON estén completos en la porción de memoria asignada a la conexión con el cliente para poder ejecutarlos.

Por ejemplo, es perfectamente posible que la aplicación reciba:

```
{
"operation":"update",
"type":"processor_state",
"id":25666,
"action":"run",
"state":"st
```

sin que esto indique un error en la transmisión de datos o un error en la forma en que el cliente construye los mensajes. La aplicación simplemente esperará a que se reciba el resto del objeto JSON para poder ejecutarlo, que en este caso tanto podría ser una actualización del estado del procesamiento del *MessageProcessor* con el identificador 25666 a *start* si termina con:

```
art”
}
```

como una actualización a *stop* si termina con:

```
op"
}
```

Qt evita al programador tener que implementar un analizador sintáctico y ofrece una API para la lectura de objetos JSON. Esta API la componen principalmente las clases:

- *QJsonDocument*: se encarga de leer y analizar sintácticamente los datos en crudo. Guarda una representación interna del objeto JSON que puede ser accedida posteriormente. También indica si ha habido error usando un objeto de la clase *QjsonParserError*.
- *QjsonParserError*: es usada para comprobar si ha habido errores en el análisis sintáctico del objeto JSON. Si ha habido error la clase indica el tipo de error y el punto de los datos en crudo donde se ha detectado el error.
- *QJsonObject*: proporciona acceso al objeto JSON guardado en una instancia de la clase *QJsonDocument*. Usando esta clase se puede acceder cómodamente desde C++ a los diferentes valores del objeto JSON.

El método *parseJson* de la clase *MainWindow* es el encargado de analizar sintácticamente los datos que se reciban provenientes de los clientes. Se compone fundamentalmente de un bucle que itera sobre la porción de memoria asignada a la conexión buscando objetos JSON completos. En cada iteración se puede dar una de estas posibilidades:

- Que encuentre el principio de un objeto JSON que no está completo. En este caso el método no hace nada, simplemente devuelve la ejecución al bucle principal de Qt.
- Que encuentre un carácter que no es el inicio de un objeto JSON, es decir un carácter que no es una llave de apertura: { . En este caso la rutina muestra un error en el registro de sucesos de la ventana principal y elimina el carácter de la memoria asignada a la conexión.
- Que encuentre un objeto JSON completo. En este caso la rutina *parseJson* llama a *doOrder* que es la encargada de despachar el objeto JSON a la rutina que ejecuta la operación en concreto que se ha recibido. Si esta ejecución termina con error, lo muestra en el registro de sucesos.

  Tras la ejecución de la orden se elimina de la porción de memoria asignada a la conexión los datos correspondientes al objeto JSON analizado.

Cuando termina la iteración del bucle la longitud de los datos contenidos en el porción de memoria de la conexión habrá disminuido, ya sea porque se haya producido un error o porque se haya ejecutado una orden. El bucle termina si la memoria asignada a la conexión está vacía.

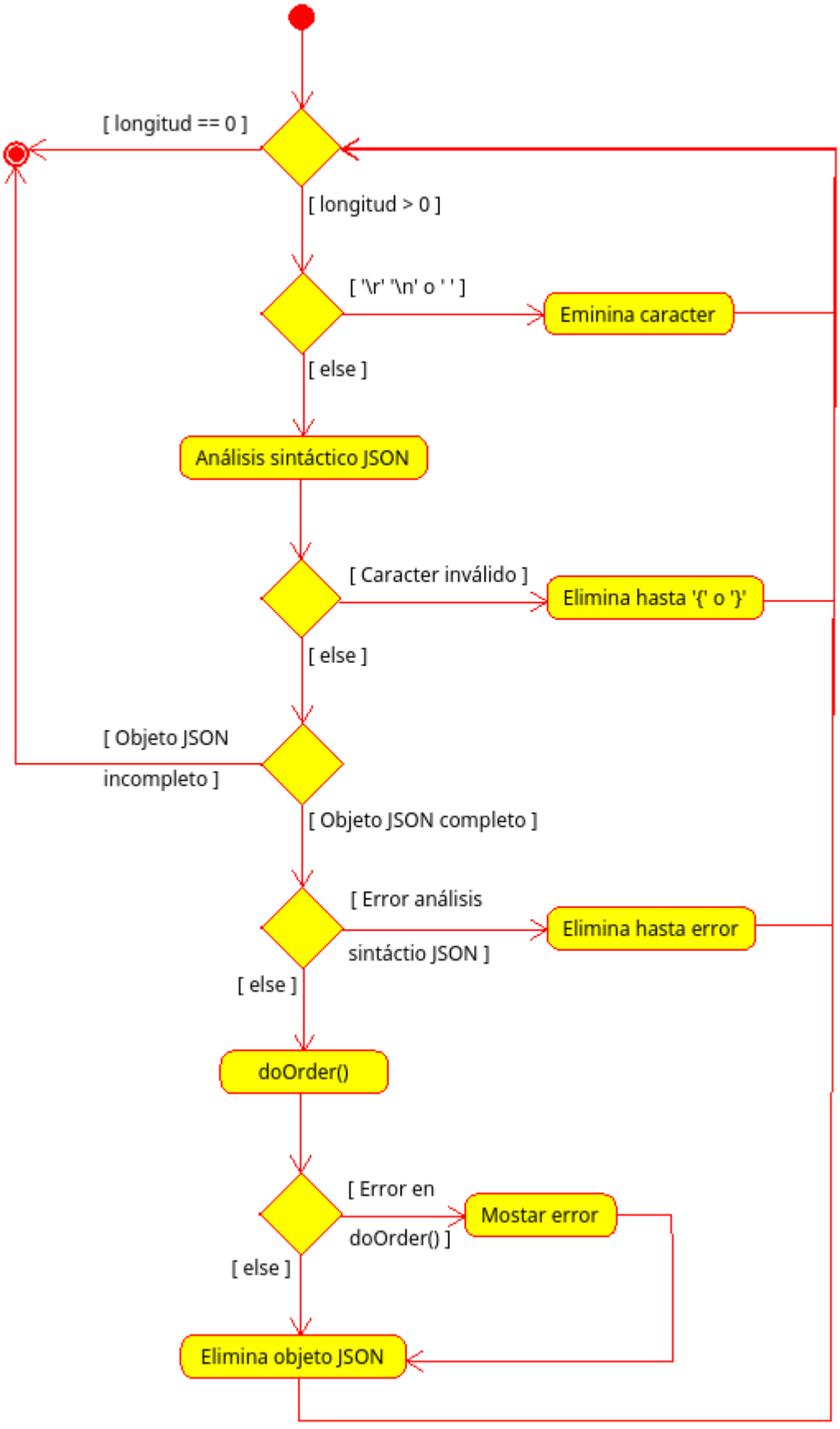

Ilustración 24: Diagrama de actividad del método parseJson()

Como se ha mencionado, la rutina *doOrder* llama a su vez a otras rutinas dependiendo del tipo de elemento sobre el que se vaya a actuar y el tipo de operación que se vaya a realizar. Estas rutinas son específicas para cada tipo de operación y son las encargadas, tras hacer las comprobaciones necesarias, de realizar las ordenes en sí mismas para que tengan reflejo en la representación gráfica.

Todas estas rutinas tienen una estructura análoga. Primero comprueban que los valores del objeto JSON son del tipo correcto, después comprueban que los elementos sobre los que se van a actuar existen o no existen dependiendo de si se van a eliminar o a crear nuevos elementos en la representación gráfica, y por último realizan la operación en sí misma.

Por ejemplo, en el caso en concreto de la rutina encargada de ejecutar la orden de eliminación de un *MessageProcessor*, llamada *doOrderDeleteProcessor*, la rutina consiste simplemente en la comprobación de la existencia en la representación gráfica de un *MessageProcessor* con el identificador que haya llegado en el objeto JSON y en la eliminación del objeto de la clase *QNEBlock* correspondiente, tras lo cual se actualiza la lista de *MessageProcessor* de la ventana principal.

4.4.2 La representación gráfica de los *MessageProcessor*

Para la representación gráfica de los *MessageProcessor* en la ventana principal se usan varias clases derivadas de las clases que Qt provee para la visualización de elementos gráficos. Estas clases, incluidas en lo que la documentación de Qt denomina *Graphics View Framework* [10], cumplen tres funciones:

- Gestión de la escena: la escena es el lienzo donde se dibujan y colocan los elementos gráficos y está implementada en la clase *QgraphicsScene*. Esta clase se encarga, entre otras cosas, de la propagación de los eventos de la interfaz gráfica a los ítems gráficos que estén contenidos en la escena.
- Presentación de una vista de la escena: la clase *QGraphicsView* es la encargada de mostrar los contenidos de una escena. Permite que el contenido de una escena pueda ser alejado y acercado y añade a la escena las barras de desplazamiento.
- Creación de elementos gráficos: Qt ofrece la clase *QGraphicsItem* como base para la creación de elementos gráficos personalizados que pueden ser añadidos a una escena. Esta clase permite que cada ítem gráfico gestione los eventos que reciba de la interfaz gráfica, además de ofrecer otras funciones.

  Basándose en *QGraphicsItem* Qt también tiene clases para formas típicas como rectángulos, círculos, textos, etc. Todas estas clases se pueden usar para la creación de formas personalizadas y pueden ser agrupadas entre sí creando nuevas formas constituidas por otras formas más simples.

A partir de las clases que Qt ofrece se han creado varias clases (*QNodesEditor*, *QNEBlock*, *QNEPort* y *QNEConnection)* especializadas en la visualización de bloques y conexiones entre bloques que representan a los *MessageProcessor* y sus relaciones.

4.4.2.1 La clase QNodesEditor

Con el objetivo de que el usuario pueda interactuar con la representación gráfica de los *MessageProcessor* en la ventana principal se crea una instancia de la clase *QGraphicsView* que presenta al usuario los elementos gráficos contenidos en una instancia de la clase *QgraphicsScene*. Esta clase, que es derivada de *QObject*, permite a otra clase, que debe ser también derivada de *QObject*, capturar sus eventos. La clase *QNodesEditor* usa esta funcionalidad para capturar los eventos dirigidos desde la interfaz gráfica a la escena y así permitir al usuario señalar, mover o eliminar los *MessageProcessor* y sus conexiones que se estén mostrando en la escena.

En el constructor de *MainWindow* se instancia la clase *QNodesEditor* y se llama al método que inicializa la captura de eventos dirigidos a la escena gráfica. También conecta las señales *updateListWidget* y *updateTableWidget* de la instancia de *QNodesEditor* con las ranuras del mismo nombre de *MainWindow*. Estas señales se usan respectivamente para actualizar los datos del *MessageProcessor* que sea seleccionado por el usuario y para actualizar la tabla con la lista de *MessageProcessor* si se elimina uno de ellos mediante la representación gráfica.

La clase *QNodesEditor* también implementa el guardado y recuperación de los *MessageProcessor* y conexiones que estén siendo mostrados en la representación gráfica. En estos métodos, la instancia de la clase *QnodesEditor* se limita a llamar a los métodos *save* y *load* de los respectivos ítems que se encuentran en la escena pasándoles como parámetro una instancia de la clase *QDataStream*. Esta instancia de la clase *QDataStream* se ha recibido de *MainWindow*, y es creada cuando se va a guardar o leer la configuración.

4.4.2.2 Las clases QNEBlock y QNEPort

Las clases *QNEBlock* y *QNEPort* implementan los elementos gráficos que representan a los *MessageProcessor* en la interfaz gráfica. La primera clase implementa el *MessageProcessor* en sí mismo y *QNEPort* implementa tanto los puntos desde los que se pueden crear conexiones entre *MessageProcessor* como el texto que hay dentro de los *MessageProcessor*. Ambas clases siempre trabajan en conjunto. Tal y como está hecha la aplicación no puede haber una instancia de *QNEBlock* sin seis instancias de *QNEPort*, ni puede haber una instancia de *QNEPort* que no esté creada por una instancia de *QNEBlock.*

Ambas clases son derivadas de *QGraphicsItem* que, como se ha mencionado anteriormente, es la clase base a partir de la cual se pueden desarrollar elementos gráficos que se pueden añadir a una escena. También ambas clases son derivadas de *QObject*, por lo que pueden usar el mecanismo señales y ranuras de Qt.

4.4.2.2.1 La clase QNEBlock

La clase *QNEBlock* puede ser instanciada por *QNodesEditor*, en el proceso de lectura de la configuración durante el arranque del sistema de monitorización, o por *MainWindow*, como resultado de una orden recibida por red.

En el primer caso, la instancia de *QNodesEditor* crea una instancia de *QNEBlock* y llama al método *load* del nuevo objeto *QNEBlock* para que lea la configuración que se guardó en una ejecución anterior de la aplicación. De esta forma todos los atributos del nuevo objeto quedan asignados con los valores correctos.

En el segundo caso, llega por red una orden con el nombre del *MessageProcessor*, el modo de gestión de los datos de entrada y el modo de gestión de los datos de salida. *MainWindow* crea el *MessageProcessor* en la representación gráfica instanciando la clase *QNEBlock* con un constructor que establece a los valores por defecto los atributos del nuevo objeto que no han llegado en la orden proveniente del *framework*.

La clase *QNEBlock* implementa, además de otros, los métodos para asignar y leer los atributos propios de un *MessageProcessor. E*stos métodos implementan la asignación o lectura de: el nombre del *MessageProcessor*, del modo de gestión de los datos de entrada, del modo de gestión de los datos de salida, del estado del procesamiento del *MessageProcessor* y la asignación o lectura de su información adicional. Existe un método para leer el identificador del *MessageProcessor*, pero no para variarlo. Esta operación no es posible ya que el identificador de los *MessageProcessor* es unívoco y no puede variar.

Los métodos de asignación de los atributos de la clase *QNEBlock* se encargan de que los cambios tengan su reflejo en la representación gráfica de los *MessageProcessor*. Por ejemplo, como se ha comentado en la descripción del protocolo de comunicaciones en el apartado 4.3.2, un *MessageProcessor* puede tener parado o arrancado el estado del procesamiento. En base a esto, el método de cambio de estado del procesamiento se encarga de que cuando un *MessageProcessor* está procesando, en el campo *Run* aparezca el tiempo que lleva transcurrido desde que se inició el procesamiento. Para implementar esto se usa la clase *QTimer*, un cronómetro de Qt, y se conecta la señal que esta emite periódicamente con una ranura que actualiza el tiempo mostrado en la representación gráfica.

4.4.2.2.2 La clase QNEPort

La clase *QNEPort* solamente es instanciada en *QNEBlock* ya que, como se ha dicho anteriormente, un puerto sólo puede existir asociado a un *MessageProcessor*, no puede existir independientemente.

Al igual que *QNEBlock*, la clase *QNEPort* ofrece, entre otros, métodos para variar la representación gráfica en base a los mensajes recibidos desde el *framework*. Estos son *SetIsStarted* y *setIsSync*.

El método *SetIsStarted* asigna a un puerto el estado arrancado o parado. Cuando la aplicación recibe

un mensaje indicando que un *MessageProcessor* ha comenzado o parado de enviar o recibir datos se llama a este método del puerto apropiado. Este método se encarga de que el color del puerto cambie en la representación gráfica dependiendo del estado del puerto. Cuando un *MessageProcessor* pasa a estar enviando su puerto de salida cambia de rojo a un parpadeo de verde y negro. Si para de enviar el puerto vuelve a rojo. De forma análoga el color del puerto de entrada cambia con el estado de la recepción del *MessageProcessor*. El parpadeo se implementa con una instancia de *QTimer* que se inicializa para que envíe una señal cada 250 milisegundos y se conecta esta señal periódica a una ranura de *QNEPort* que cambia el color del puerto.

El método *setIsSync* es llamado por la instancia de QNEBlock a la que pertenece el puerto cuando la aplicación recibe una orden indicando que el modo de gestión de la entrada o la salida del *MessageProcessor* que representa el bloque ha cambiado. Este método se encarga de que la linea que representa la conexión entre dos *MessageProcessor* sea continua si los modos de gestión de los puertos de salida y entrada de la conexión son síncronos. En cualquier otro caso la linea mostrada es discontinua. Para realizar esto el método de *QNEPort* llama al método *setIsAsync* de *QNEConnection*.

4.4.2.3 La clase QNEConnection

Esta clase implementa la representación gráfica de las conexiones entre *MessageProcessor*. La mayor parte del código implementa la gestión de las conexiones en sí y la representación gráfica de los enlaces entre bloques. Con respecto a las funcionalidades propias de la aplicación desarrollada sólo es destacable el método *setIsAsync* que es al que llaman las instancias de la clase *QNEPort* cuando se cambia el modo de gestión de los datos de entrada o salida de un *MessageProcessor*.

Como se ha comentado en el apartado anterior, si ambos extremos de la conexión tienen una gestión de datos síncrona en la representación gráfica aparece una linea continua. En cualquier otro caso la linea es discontinua.

5 Aplicación sobre un entorno de vigilancia marítima

Como ejemplo práctico de uso del *framework* descrito en el capítulo 3 se ha desarrollado un sistema de fusión de datos que a su vez ha sido usado para el desarrollo de un sistema de vigilancia marítima [11].

Los sistemas de vigilancia marítima utilizan como fuente de datos la información proporcionada por los radares costeros y por los AIS.

Los radares son capaces de monitorizar grandes superficies y en un entorno marítimo, como la vigilancia costera, pueden generar una gran cantidad de información debido a la densidad de blancos detectables.

Los AIS son transpondedores ubicados en determinadas embarcaciones que proporcionan su ubicación en tiempo real junto con algunas características de la embarcación como dimensiones, velocidad, rumbo, etc.

En base a estos datos, el sistema desarrollado usando el *framework* es capaz de realizar en tiempo real las transformaciones adecuadas a los datos de las distintas fuentes para presentar en la pantalla de un operador una información unificada y precisa de cada blanco.

Al ser fuentes de datos independientes es posible que del mismo blanco lleguen datos reiterados o incluso datos divergentes. Por esto es necesario realizar una fusión de los datos provenientes de las diferentes fuentes con el fin de evitar que las discrepancias en las entradas generen salidas erróneas.

El problema consiste en realizar el tratamiento de los datos recibidos de los radares y de los AIS, y presentar una imagen precisa y actualizada de todas las naves que se encuentren en la zona de cobertura. El software se encarga de que los barcos estén localizables de forma continua y en tiempo real independientemente de los datos ocasionalmente erróneos que faciliten los sensores y de los fallos que puedan producirse en estos mismos sensores o en otros mecanismos intermedios.

Este proceso de transformación y fusión de los datos es realizado, usando el *framework*, por unidades de procesamiento encargadas cada una de un paso en concreto:

- Recepción y transformación de la señal: el sistema transforma los datos a coordenadas cartesianas para facilitar su posterior tratamiento.
- Asociación de las señales por sensor: se asocian las detecciones de blancos recibidas a posibles trayectorias existentes en el sistema o se crean nuevas trayectorias si fuese necesario.
- Asociación de las señales de todos los sensores: por último se fusionan todas las trayectorias detectadas en cada uno de los sensores.

Ilustración 25: Esquema del uso del framework por el sistema de vigilancia marítima

Como los dos primeros pasos se pueden realizar en paralelo, el sistema se pude representar esquemáticamente de la siguiente manera:

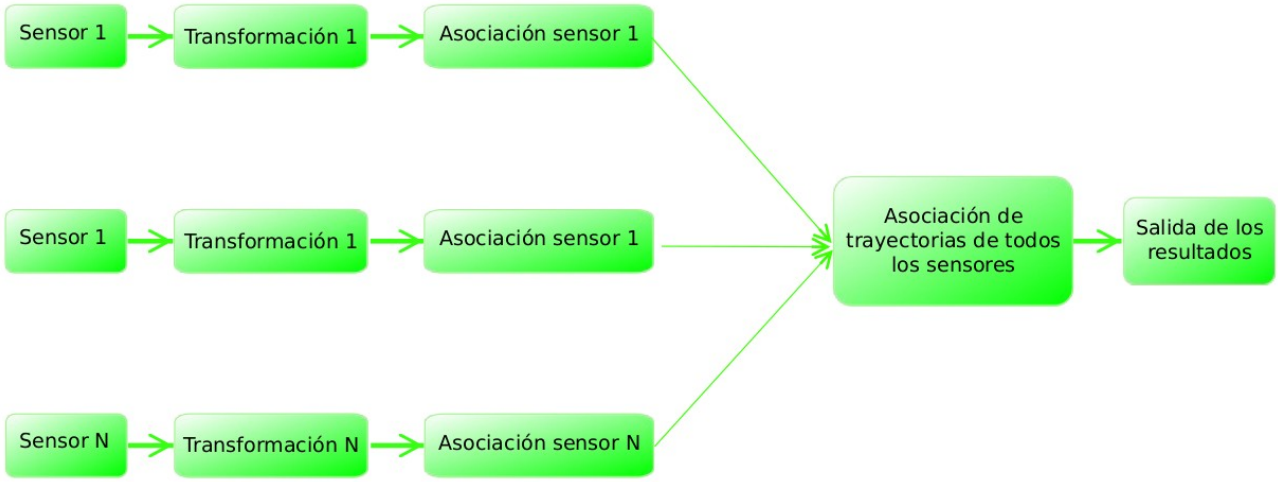

Ilustración 26: Esquema del uso del framework con MessageProcessor en paralelo

En un uso real el número de sensores es cuantioso y el trasiego de información entre las diferentes fases del sistema es continuo, lo que crea un elevado nivel de dinamismo durante las ejecuciones.

En la ilustración 27 se muestra una representación gráfica del sistema de vigilancia marítima.

Ilustración 27: Representación del sistema de vigilancia marítima.

6 Conclusiones y trabajos futuros

Como ha quedado expuesto a lo largo de la presente memoria, los desarrollos realizados usando el *framework* para el procesado de información pueden alcanzar una complejidad muy considerable. Adicionalmente, durante la ejecución de estos desarrollos las distintas unidades de procesamiento están intercambiando información entre ellas de forma continua. Los envíos y recepciones se realizan en base a criterios específicos de cada unidad de procesamiento y, además, el estado de cada una de estas unidades varía a lo largo del tiempo dependiendo de varios factores que en conjunto no son predecibles de forma sencilla.

Para facilitar la visualización del funcionamiento interno de las aplicaciones desarrolladas usando el *framework* para el procesado de información se decidió desarrollar un sistema de monitorización que mostrase visualmente y en tiempo real las transiciones por las que evolucionan las diferentes unidades de procesado del *framework* durante su ejecución.

El que el *framework* sea un desarrollo multiplataforma ha implicado que los requerimientos principales para el sistema de monitorización fueran que igualmente fuese multiplataforma y que se comunicase con el *framework* utilizando protocolos normalizados.

En base a estos requerimientos, el desarrollo del sistema de monitorización se ha realizado usando C++ como lenguaje de programación con la librería Qt como principal apoyo, lo que ha permitido que al escribir el código no se tuviese que tener preocupación alguna por la portabilidad entre plataformas, ya que la librería Qt oculta totalmente los diferentes detalles de cada sistema operativo y provee de una API que es común a todos ellos para realizar todo tipo de operaciones.

Para la comunicación con el *framework* se ha usado TCP como protocolo de nivel de transporte y JSON como protocolo de nivel de aplicación. Ambos protocolos están ampliamente difundidos, lo que posibilita la comunicación con aplicaciones que se ejecuten en diferentes plataformas y que el sistema de monitorización en sí mismo sea muy portable.

Con el objetivo de evaluar el sistema de monitorización se ha aplicado sobre un entorno de vigilancia marítima que ha sido desarrollado usando el *framework* de procesado de información. Se ha podido comprobar su validez y utilidad para la visualización de la evolución de las unidades de procesamiento así como para comprobar si el flujo de información era el esperado.

Las futuras líneas de trabajo sobre el desarrollo realizado son múltiples. Destacan:

- Mejoras en la representación gráfica del *framework*:
    - Realización de mejoras en la representación gráfica de los *MessageProcessor* como la incorporación de gráficos animados que representaran los diferentes modos de gestión

de la entrada, de la salida y el modo de procesamiento, o dar al usuario la posibilidad de elegir entre diferentes esquemas de colores.

- Ofrecer al usuario la posibilidad de deshacer los movimientos realizados en la representación gráfica, permitiendo de esta manera realizar cambios con la posibilidad de poder volver a la situación inicial si el resultado no es el deseado.
- Guardar un histórico de los cambios realizados en la representación gráfica, ya sean los realizados por el usuario o los realizados por la aplicación en base a las órdenes recibidas desde el *framework*. Una vez desarrollada esta nueva funcionalidad se podría ofrecer al usuario la posibilidad de hacer una reproducción en diferido de la evolución de la escena completa, permitiendo ver la secuencia de pasos que se han producido hasta llegar a la situación actual. La reproducción de la secuencia de pasos sería controlable por el usuario de forma que pudiese avanzar o retroceder a la velocidad deseada. La secuencia de pasos podría ser guarda en disco para permitir un análisis posterior.

- Portabilidad a otras plataformas, como Android o iOS para permitir la ejecución de la aplicación desarrollada en dispositivos móviles. Estas plataformas están soportadas por Qt por lo que la compilación no debería presentar problemas. No ocurre lo mismo con la distribución del paquete de software.

  En el caso de Android existe la posibilidad de abrir una cuenta de desarrollador en Google Play y, tras el correspondiente pago por el alta de la cuenta, se podría subir la aplicación y los usuarios podrían descargarla e instalarla en sus dispositivos. Android también permite la instalación de aplicaciones no firmadas, por lo que la aplicación podría ser distribuida usando algún método alternativo a Google Play como el correo electrónico, un enlace de descarga o una conexión *Bluetooth.* El usuario debe activar la opción “Orígenes desconocidos” dentro del menú relativo a la seguridad para permitir este tipo de instalaciones.

  En el caso de iOS sólo existe la posibilidad de distribuir aplicaciones a través de App Store tras abrir una cuenta en el *iOS Developer Program* y realizar el correspondiente pago con periodicidad anual.

- Nuevas funcionalidades como:
  - Ampliar las órdenes provenientes del *framework*. El variar el protocolo de comunicación implicaría cambios en la rutina *doOrder* que se encarga de la selección del método a ejecutar dependiendo de la orden recibida y, por supuesto, conllevaría la implementación de las nuevas órdenes. Es posible que se tuvieran que modificar las clases gráficas para poder mostrar en la representación gráfica las nuevas características o nuevos estados de los *MessageProcessor* .

- Permitir monitorizar activamente los diferentes *MessageProcessor*. El sistema de monitorización permitiría crear alarmas compuestas por unas condiciones y unas acciones asociadas. En el momento que se cumpliesen las condiciones de la alarma se ejecutarían las acciones.

  Las condiciones podrían estar basadas en umbrales, como tiempo sin transmitir o recibir, tiempo procesando o podrían estar basadas en otros baremos como reiterados cambios de estado.

  Las acciones asociadas a la alarma podrían abarcar un amplio rango como mensajes por pantalla, correo electrónico, notificaciones en el móvil... acciones sobre el *framework* o acciones sobre algún elemento que proporcione entradas al *framework*.

- Permitir al usuario del sistema de monitorización variar el funcionamiento del *framework* usando como interfaz la representación gráfica de los *MessageProcessor*. Esta variación del sistema de monitorización consistiría en añadir a la representación gráfica la capacidad para que el usuario pudiera modificar conexiones entre los *MessageProcessor* o alguna característica de estos. Así mismo el protocolo de comunicaciones debería ser modificado para permitir al sistema de monitorización informar al *framework* de los cambios que debe realizar en base a las operaciones realizadas por el usuario. Por último, habría que implementar las operaciones en el *framework*.

7 Anexos

7.1 Software entregado

En este anexo se describe el proceso de creación de los paquetes de software para usuarios finales que se entregan para los diferentes sistemas operativos en los que se puede ejecutar la aplicación desarrollada. Aunque Qt proporciona una vía para la creación de los binarios, cada sistema operativo tiene sus peculiaridades en cuanto a la creación de un paquete listo para la distribución a usuarios finales.

7.1.1 OS X

La creación de un paquete para la instalación de la aplicación desarrollada para este sistema operativo ha sido sumamente sencilla. La compilación se puede llevar a cabo una vez estén instaladas las utilidades de desarrollo que proporciona Apple para OS X, que están incluidas en el paquete *Xcode* que se puede instalar desde *Mac App Store*, y el entorno de desarrollo *Qt Creator* de Qt, que se puede bajar desde la propia página de Qt.

En la compilación, si se dejan las opciones por defecto, se crea un directorio con el nombre *build-Rgrafica-Desktop_Qt_5_3_1_clang_64bit-Release* en el que se almacenan los ficheros objeto y el ejecutable resultante, en este caso *Rgrafica*. Una vez se tiene el ejecutable se crea el paquete lanzando desde una terminal la instrucción *macdeployqt* que está incluida en *Qt Creator*. La orden debe ser algo similar a:

```
$ /Users/usuario/Qt5.3.1/5.3/clang_64/bin/macdeployqt /Users/usuario/Desktop/build-Rgrafica-
Desktop_Qt_5_3_1_clang_64bit-Release/Rgrafica.app -dmg
```

El fichero *Rgrafica.dmg* resultante, que se genera en el mismo directorio, es el que puede ser distribuido.

Para instalar la aplicación basta con abrir el archivo *Rgrafica.dmg* y arrastrar la aplicación a la carpeta de aplicaciones, que previamente se debe haber abierto en otra ventana de *Finder*.

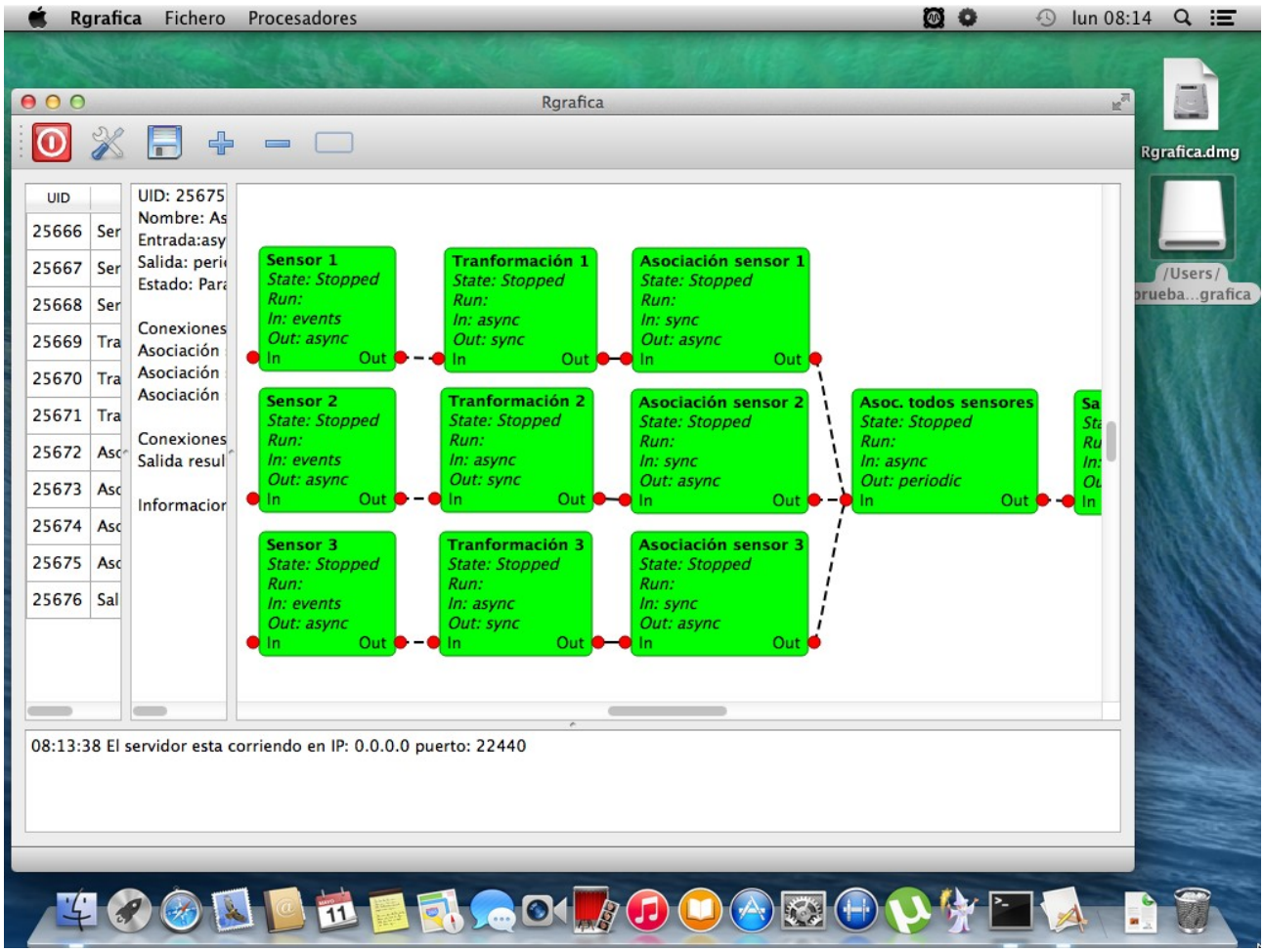

Ilustración 28: La aplicación desarrollada corriendo en OS X Mavericks 10.9

7.1.2 MS-Windows

Para crear el paquete de software para usuarios finales en MS-Windows hay que instalar el entorno de desarrollo *Qt Creator* que está disponible en la página de descargas de Qt. Se ha optado por descargar la versión que viene con el compilador *minGW* por ser gratuito.

Una vez instalado el entorno de desarrollo con el compilador y generado el ejecutable se ha de hacer una copia del ejecutable y de las DLL del compilador a un directorio nuevo. Las órdenes a ejecutar deben ser similares a:

```
C:\> copy Rgrafica.exe C:\Rgrafica
C:\> copy C:\Qt\Qt5.3.1\5.3\mingw482_32\bin\libwinpthread-1.dll C:\Rgrafica
C:\> copy C:\Qt\Qt5.3.1\5.3\mingw482_32\bin\libgcc_s_dw2-1.dll C:\Rgrafica
C:\> copy C:\Qt\Qt5.3.1\5.3\mingw482_32\bin\libstdc++-6.dll  C:\Rgrafica
```

Hecho esto se ha de ejecutar la orden *windeployqt* que está incluida en Qt Creator:

```
C:\> set PATH=%PATH%;C:\Qt\Qt5.3.1\5.3\mingw482_32\bin
C:\> windeployqt C:\Rgrafica
```

Con este comando se consigue tener en el directorio destino, `C:\Rgrafica` en los ejemplos, un ejecutable funcional que puede ser distribuido si se empaqueta con cualquier compresor, como por ejemplo *WinZip*.

La práctica común en los entornos de escritorio de MS-Windows es que las aplicaciones se distribuyan en ficheros ejecutables auto-extraibles que se encargan de realizar la copia de los ficheros de la aplicación a los directorios dedicados a ello por el sistema operativo e igualmente se encargan de crear las entradas de menú en el escritorio del usuario.

Microsoft proporciona la herramienta *Windows Installer* para crear estos ejecutables auto-extraibles pero su uso, aunque no es complicado, es extremadamente laborioso puesto que abarca todas los posibilidades que se pueden dar a la hora de distribuir públicamente una aplicación. Por ello se ha usado la herramienta *Advanced Installer*, que permite crear los ejecutables de instalación de una forma más sencilla si no se tienen requerimientos especiales.

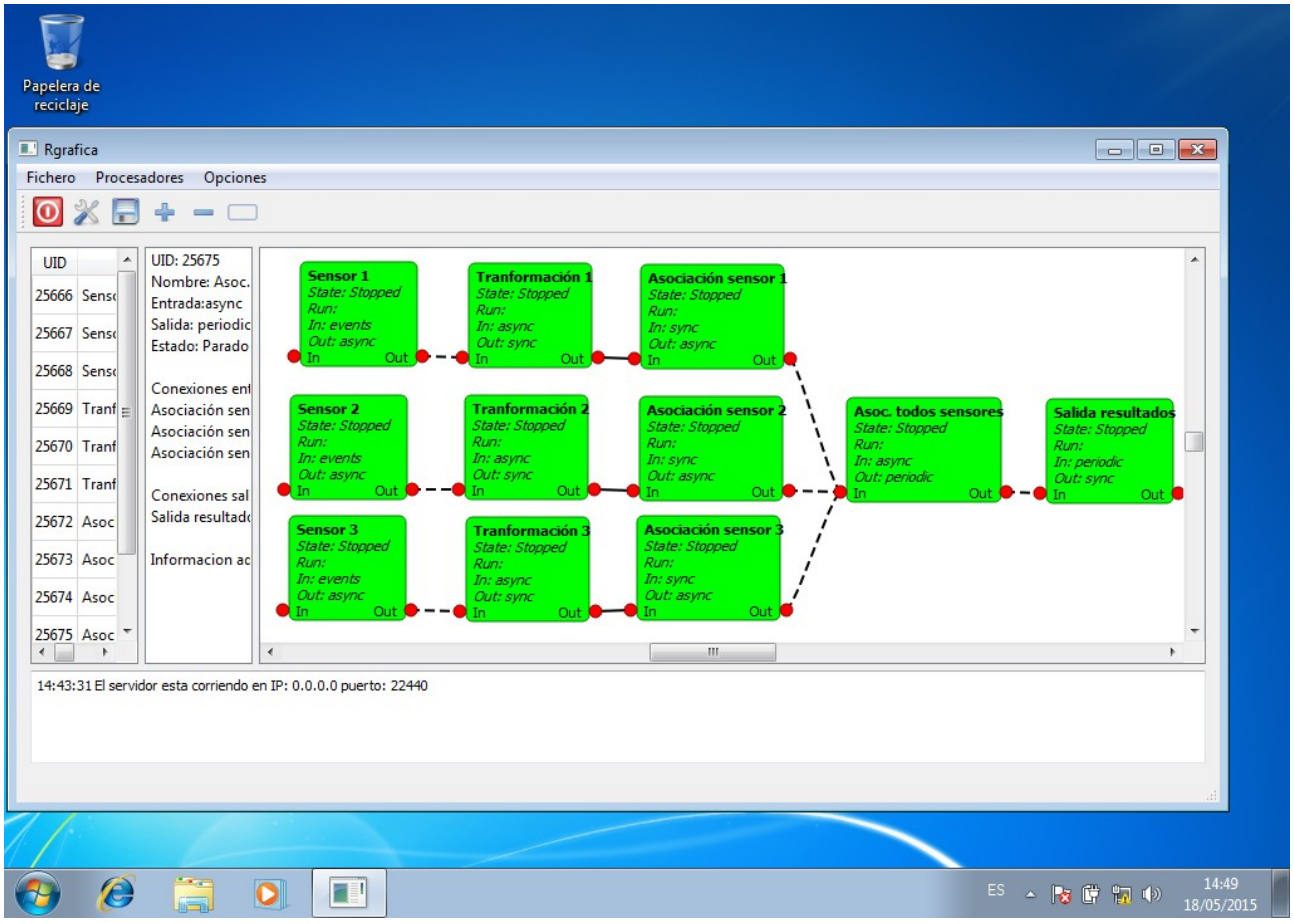

Ilustración 29: La aplicación desarrollada corriendo en MS-Windows 7

7.1.3 Linux

La heterogeneidad de las múltiples distribuciones Linux ha hecho inviable la entrega de un paquete de software para cada una de ellas. Se ha optado por entregar el paquete para Ubuntu por ser la más extendida en el escritorio. Se ha creado un paquete que puede ser instalado en todas las actualizaciones que han aparecido desde la última versión con un periodo de soporte extendido hasta el momento actual, es decir el paquete es instalable en Ubuntu 14.04 LTS, 14.10 y 15.04.

Para la creación del paquete, al igual que en los otros sistemas operativos, primero se ha instalado el compilador de C++, en este caso *g++* que está disponible en los propios repositorios de Ubuntu, y el entorno de desarrollo *Qt Creator*. Un vez se ha compilado el código fuente se elimina la información de depuración, que no tiene valor para el usuario final, y se copia al directorio en el que se quiere que lo instalen los usuarios finales:

```
$ strip Rgrafica
$ sudo cp Rgrafiga /usr/bin
```

Después se crea un archivo para la entrada en el menú de aplicaciones con el contenido:

```
[Desktop Entry]
Type=Application
Exec=/usr/bin/Rgrafica
Name=Rgrafica
GenericName=Representación gráfica de MessageProcessors
Terminal=false
Categories=Utility
```

y se copia en el directorio adecuado:

```
$ sudo cp Rgrafica.desktop /usr/share/applications
```

Para crear el paquete se ha utilizado la herramienta *fpm* que se instala con:

```
$ sudo apt-get install ruby ruby2.1-dev
$ sudo gem install fpm
```

Finalmente se crea el paquete con la orden:

```
$ fpm -s dir -t deb -n rgrafica -v 1.0 \
  -d "libqt5core5a >= 5.3.1" -d "libqt5gui5 >= 5.3.1" \
  -d "libqt5network5 >= 5.3.1" \
  /usr/bin/Rgrafica /usr/share/applications/Rgrafica.desktop
```

Esto da como resultado un paquete con el nombre *rgrafica_1.0_amd64.deb* que puede ser instalado con la orden:

```
$ sudo dpkg -i rgrafica_1.0_amd64.deb
```

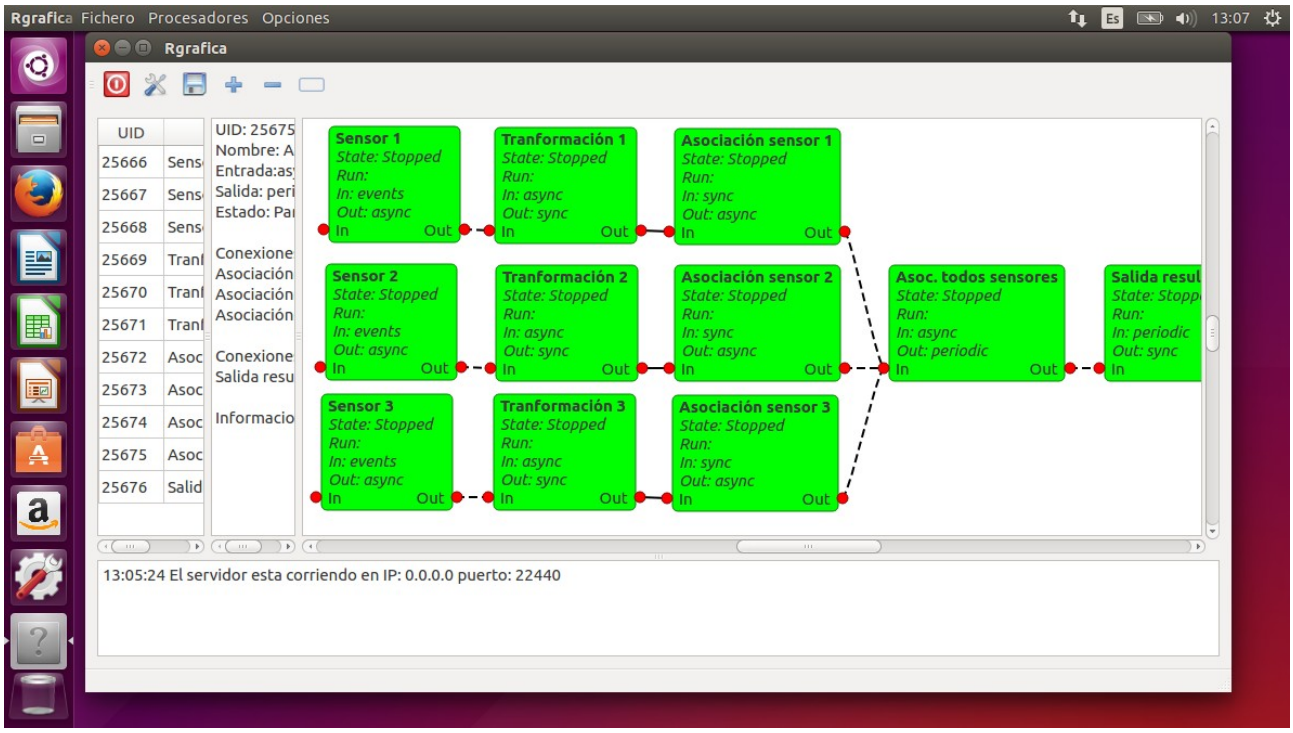

Ilustración 30: La aplicación desarrollada corriendo en Ubuntu 15.04

en cualquier sistema Linux con arquitectura X86_64 que esté basado en paquetes en formato *.deb* y que cumpla las dependencias que se han indicado en la creación del paquete, esto es que tenga previamente instalado las librerías Qt versión 5.2.0 o superior.

7.2 Glosario

AIS Acrónimo de *Automatic Identification System*, sistema de identificación automática, pudiéndose encontrar como SIA. El objetivo fundamental del sistema AIS es permitir a los buques comunicar su posición y otras informaciones relevantes para que otros buques o estaciones puedan conocerla y evitar colisiones.

API Acrónimo de *Application Programming Interface*, interfaz de programación de aplicaciones. Es el conjunto de subrutinas, funciones y procedimientos (o métodos, en la programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción. Son usadas generalmente en las bibliotecas.

DLL Acrónimo de D*ynamic-Link Library*, biblioteca de enlace dinámico. Es el término con el que se nombra a los archivos con código ejecutable que se cargan bajo demanda de un programa por parte de los sistemas operativos de la familia MS-Windows .

EBNF Acrónimo de *Extended Backus–Naur Form*, es un metalenguaje usado para expresar gramáticas libres de contexto: es decir, una manera formal de describir lenguajes formales, como son los lenguajes de programación.

ERP Acrónimo de *Enterprise Resource Planning*, planificación de recursos empresariales. Es un

software de gestión del negocio, típicamente una *suite* de aplicaciones integradas que una compañía usa para recolectar, almacenar, gestionar e interpretar datos de diferentes procesos del negocio. Normalmente cubren la planificación de costes, el reparto, las ventas, la gestión de inventario...

Framework Conjunto estandarizado de conceptos, prácticas y criterios para enfocar un tipo de problemática particular que sirve como referencia, para enfrentar y resolver nuevos problemas de índole similar.

JSON Acrónimo de *JavaScript Object Notation*, es un formato de estándar abierto que usa texto legible para un humano para transmitir datos consistentes en parejas atributo-valor. Es usado principalmente para transmitir datos entre un servidor y una aplicación web, como una alternativa a XML.

KDE Acrónimo de *K Desktop Enviroment*, es un entorno de escritorio y una serie de librerías y aplicaciones desarrolladas con las librerías Qt.

PLC Acrónimo de *Programmable Logic Controller*, controlador lógico programable. Es una computadora son un sistema en tiempo real utilizada para automatizar procesos electromecánicos.

POSIX Acrónimo de *Portable Operating System Interface for uniX*, interfaz portable de sistema operativo para Unix. Es una norma [12] escrita por la IEEE que define una interfaz estándar del sistema operativo y el entorno, incluyendo un intérprete de comandos y programas de utilidades comunes para apoyar la portabilidad de las aplicaciones a nivel de código fuente.

SCADA Acrónimo de *Supervisory Control And Data Adquisition,* supervisión, control y adquisición de datos. Es un software para ordenadores que permite controlar y supervisar procesos industriales a distancia.

SNMP Acrónimo de *Simple Network Management Protocol*, protocolo simple de administración de red. Es un protocolo de nivel de aplicación para facilitar el intercambio de información de administración entre dispositivos de red.

Socket Abstracción, generalmente proporcionada por el sistema operativo, que permite a un proceso enviar información a través de la red a otro proceso independientemente de si se encuentran en en mismo ordenador. Un *socket* queda definido cuando se especifican las direcciones IP y puertos de cada extremo y el protocolo de transporte. Con esta información los sistemas operativos de ambos extremos pueden entregar los datos intercambiados a los procesos correspondientes.

TCP Acrónimo de *Transmission Control Protocol*, protocolo de control de transmisión. Es uno de los protocolos fundamentales en Internet. Este protocolo garantiza que los datos serán entregados en su destino sin errores y en el mismo orden en que se transmitieron. También proporciona un mecanismo para distinguir distintas aplicaciones dentro de una misma

máquina, a través del concepto de puerto.

RFC Acrónimo de *Requests for Comments*, petición de comentarios. Son los documentos en los que se detallan los estándares de los protocolos y otros aspectos del funcionamiento de internet y otras redes.

Unicode Es un estándar de codificación de caracteres diseñado para el tratamiento informático, transmisión y visualización de textos de múltiples lenguajes.

XML Acrónimo de *eXtensible Markup Language*, lenguaje de marcas extensible. Es un lenguaje de marcas desarrollado por el World Wide Web Consortium utilizado para almacenar datos en forma legible. Se propone como un estándar para el intercambio de información estructurada entre diferentes plataformas.

7.3 Bibliografía

[1] I. Sommerville, en "*Sistemas críticos*" y "*Desarrollo de sistemas críticos*", en *Ingeniería del Software*. 7ª ed. Pearson Educación, 2005. ISBN: 8478290745, pp. 39-58 y pp. 423-446

[2] B. González Pérez. *Diseño de un framework de procesamiento de información y su aplicación real a la fusión de datos*. Trabajo fin de grado. Colmenarejo. Universidad Carlos III de Madrid, 2013. Disponible en: http://e-archivo.uc3m.es/handle/10016/19147 [consulta: 25 Junio 2015]

[3] Instituto Nacional de Estadística. *Encuesta trimestral de coste laboral (1/2015)*. Disponible en: http://www.ine.es/dyngs/INEbase/es/operacion.htm?c=Estadistica_C&cid=1254736045053&menu=ultiDatos&idp=1254735976596 [Consulta: Julio 2015]

[4] España 2014. BOE-A-2014-12328. *Ley 27/2014, de 27 de noviembre, del Impuesto sobre Sociedades. Boletín oficial del estado*. Cap 2. Disponible en: http://boe.es/buscar/act.php?id=BOE-A-2014-12328 [Consulta: Junio 2015]

[5] RFC 7159: The JavaScript Object Notation (JSON) Data Interchange Format. Internet Engineering Task Force. Disponible en: http://tools.ietf.org/html/rfc7159 [Consulta: Junio 2015]

[6] *Boost C++ Libraries*. Disponible en: http://www.boost.org [Consulta: Junio 2015]

[7] "Qbs Manual", *Qt Documentation*, Disponible en: http://doc.qt.io/qbs [Consulta: Junio 2015]

[8] The Open Group Base Specifications Issue 7, IEEE Std 1003.1, 2013 Edition. Disponible en: http://pubs.opengroup.org/onlinepubs/9699919799/functions/listen.html [Consulta: Junio 2015]

[9] The Open Group Base Specifications Issue 7, IEEE Std 1003.1, 2013 Edition. Disponible en: http://pubs.opengroup.org/onlinepubs/9699919799/functions/accept.html [Consulta: Junio 2015]

[10] "Graphics View Framework", *Qt Documentation*, Disponible en: http://doc.qt.io/qt-

5/graphicsview.html [Consulta: Junio 2015]

[11] J. García, J. L. Guerrero, A. Luis y J. M. Molina. “Robust Sensor Fusion in Real Maritime Surveillance”. Presentado a: *Proceedings of the 13th International Conference on Information Fusion*, Edinburgo, Reino Unido, 26-29 Julio 2010. Disponible en: http://e-archivo.uc3m.es/bitstream/10016/9326/3/robust_molina_Fusion10_2010.pdf [consulta: 15 Noviembre 1014]

[12] The Open Group Base Specifications Issue 7, IEEE Std 1003.1, 2013 Edition. Disponible en: http://pubs.opengroup.org/onlinepubs/9699919799/ [Consulta: Junio 2015]

7.4 Otros recursos

A lo largo del desarrollo y documentación del presente proyecto se han utilizado los siguientes recursos:

- La librerías Qt y el entorno de desarrollo *Qt Creator* han sido usados para la creación de la aplicación y pueden ser descargados de http://www.qt.io/download. La versión de Qt usada ha sido la *Community*.
- Para la implementación de la representación gráfica de los *MessageProcessor* se ha tomado como base la librería disponible en http://nukengine.com/qt-node-editor
- El presente documento ha sido elaborado usando LibreOffice Writer, el procesador de texto de la suite ofimática LibreOffice que está disponible en http://es.libreoffice.org
- Para la generación de los diagramas UML se ha usado Umbrello que está disponible en https://umbrello.kde.org
- Para la generación de los diagramas Gantt se ha usado GanttProject que está disponible en http://www.ganttproject.biz
- Para la generación de diagramas sintácticos a partir de EBNF se ha usado: http://bottlecaps.de/rr/ui
- El resto de las ilustraciones han sido realizadas con Calligra Flow que se encuentra disponible en https://www.calligra.org/flow
- Para la generación del paquete para MS-Windows se ha usado Advanced Installer que está disponible en http://www.advancedinstaller.com
- Para la generación del paquete para Ubuntu Linux se ha usado Effing Package Management que está disponible en https://github.com/jordansissel/fpm